UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Desarrollo de un sistema de SLAM para el robot AR.Drone en entorno ROS"

Álvaro Andrés Saltos Vásquez.

2015

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Desarrollo de un sistema de SLAM para el robot AR.Drone en entorno ROS"

Alumno: Álvaro Andrés Saltos Vásquez.

Director: Dña. Mª Elena López Guillén.

Tribunal:

Presidente: D. Rafael Barea Navarro.

Vocal 1º: D. Javier Macías Guarasa.

Vocal 2º: Dña. Mª Elena López Guillén.

Calificación: .....

Fecha: .....

Lo que importa verdaderamente en la vida no son los objetivos que nos marcamos, sino los caminos que seguimos para lograrlo.

-Peter Bamm-

Vale más actuar exponiéndose a arrepentirse de ello, que arrepentirse de no haber hecho nada.

-Boccaccio, Giovani-

Agradecimientos

En primer lugar, quiero agredecer a mi tutora y profesora Elena que durante todo este año ha estado trabajando a mi lado y siempre ha tenido un rato para resolver mis dudas y ayudarme en todo lo que ha podido. Para continuar, agradezco a mi familia, a mi padre Álvaro, a mi madre Silvia y a mi hermana Diana, que desde que empecé esta carrera siempre me han apoyado y ayudado, y sin ellos nada de esto fuese posible. Para terminar, quiero agradecer a una persona que durante este año ha estado a mi lado en las buenas y en las malas, apoyándome en mis momentos de decaída y animándome a redactar este proyecto, que en ocasiones ha sido difícil, y por ello se merece estos agradecimientos, gracias Marta. Ah, y no me olvido de mi compañera de juego, mi perra Troya.

Índice general

Índice de figuras

Índice de tablas

Resumen

En el presente documento se describe el proceso llevado a cabo para desarrollar un sistema de localización y mapeado simultáneo (SLAM) para el cuadricóptero AR Drone 2.0 de Parrot. Para ello, se abarcan diferentes objetivos a lo largo del proyecto tales como conocer el entorno de programación robótico en el que se trabajará, en este caso ROS, desarrollar un simulador que permita realizar pruebas iniciales sobre un modelo del robot, implementar un sistema de mapeado 2D basado en medidas de un sensor láser incorporado al mismo, y finalmente diseñar un estimador de posición utilizando tanto las medidas del láser, como las proporcionadas por el resto de sensores a bordo de la plataforma robótica.

Palabras clave: cuadricóptero, SLAM, ROS.

Abstract

In this document it is described the process performed to make a localization and mapping system (SLAM) for Parrot's AR Drone 2.0. To do this, different goals are marked along this project, such as know the robotic development environment, in this case ROS, make a simulator which allows do initials proofs with a robot model, implement a mapping 2D system based in measurements from a laser sensor incorporated, and finally design a position 3D estimator using laser measurements, as provided by other sensors on the robot platform.

Key words: drone, SLAM, ROS.

Resumen extendido

El uso de plataformas robóticas hoy en día es tan cotidiano que algunas de ellas son imprescindibles en muchas aplicaciones, desde la fabricación de automóviles hasta la exploración de terrenos desconocidos. Es precisamente en estas tareas de reconocimiento de terreno el uso justificado de robots ya que ha provocado que exista un avance tecnológico en la investigación de nuevas técnicas que mejoren cada vez más el reconocimiento de entornos mediante el uso de estas plataformas.

Es conocido el uso de vehículos terrestres para la obtención de mapas mediante el uso de las técnicas SLAM[1]. Estos vehículos en ocasiones no proporcionan toda la información del terreno que es necesaria ya que existen zonas inaccesibles para ellos, por lo que ha crecido el uso de vehículos aéreos para la obtención de mapas usando diferentes técnicas de SLAM.

El presente documento describe el proceso llevado a cabo para el desarrollo de un sistema SLAM para el robot AR Drone 2.0 basado en el algoritmo Hector-Mapping. Para ello, se ha utilizado el entorno de desarrollo robótico ROS (Robot Operating System) el cual proporciona una serie de herramientas robóticas y drivers que permiten el control y lectura de datos de diferentes sensores.

La simulación de este tipo de proyectos es necesaria ya que permite conocer algunas características y posibles futuros errores con el robot real. Para la simulación se utilizará un programa llamado Gazebo, en el que se cargará un modelo 3D previamente creado con un programa de diseño 3D, en este caso SolidWorks, y se utilizará un entorno de un edificio para la simulación del SLAM. Para la visualización del mapa 2D obtenido, se utilizará Rviz que es un programa de visualización robótica.

Se usarán paquetes de drivers que serán instalados en ROS para el control del AR Drone y la lectura de los diferentes sensores que éste tiene en su estructura. Una vez obtenido el entorno de simulación y probada la lectura de los diferentes sensores del robot real, se montará la plataforma que será utilizada para este proyecto. En esta plataforma real se incluye un sensor láser Hokuyo URG-04LX y una Raspberry Pi, la cuál permitirá el envío de datos del láser al ordenador portatil en el que estarán instalados todos los alogritmos necesarios para el SLAM. Cuando esté correctamente montada la estructura, se realizarán las diferentes pruebas y la lectura de datos para la obtención del mapa 2D en entornos reales.

Una de las características del SLAM es la localización del robot en el mapa obtenido del entorno. Se ha creado un algoritmo de estimación de posición 3D mediante el programa matemático MATLAB. Con este estimador se pueden realizar pruebas mediante la introducción de datos arbitrarios, es decir, introducción de diferentes configuraciones de control simulando el vuelo del AR Drone en MATLAB para obtener una trayectoria estimada del robot y, a su vez, se puede obtener la estimación de posición del robot real a través de los datos de los diferentes sensores del robot.

[1] siglas en inglés Simultaneous Localization and Mapping o en español localización y mapeado simultáneo

Lo anteriormente descrito se ha estructurado en diferentes capítulos en los que se detallan los pasos que se han llevado a cabo para la creación de este proyecto.

MEMORIA

Capítulo 1

Introducción

La creciente investigación sobre MAVs[1], también conococidos como pequeños vehículos aéreos, y la consiguiente mejora de las tecnologías implicadas, como el avance en el estudio y creación de nuevos y mejores microcomputadores y dispositivos sensoriales, han aumentado los requisitos de rendimiento de este tipo de sistemas. Si a este avance se le añaden las diversas aplicaciones que tienen estas plataformas robóticas, se abre un gran abanico de diversos proyectos con estos dispositivos. En la figura 1.1 se pueden observar unos ejemplos de aplicaciones con MAVs hoy en día.

(a) Rescate con drones. (b) Fotografía aérea.

(c) Drones para cultivos.

Figura 1.1: Aplicaciones con drones.

La posibilidad de introducir dispositivos GPS en los MAVs y gracias a los sensores inerciales que llevan, les permiten realizar vuelos en exteriores sin la intervención humana [1], [2] y [3]. Desafortunadamente, la mayoría de los entornos de interior aún permanecen sin acceso a los

[1] siglas en inglés de micro aereal vehicle

sistemas de localización externos y los MAVs autónomos son muy limitados en su capacidad para operar en estas áreas.

Para la localización y mapeado simultáneo (SLAM) mediante el uso de vehículos terrestres no tripulados el uso de GPS y sensores acoplados a los vehículos, permiten obtener mejores resultados de mapas del entorno mediante el uso de técnicas de SLAM y el trabajo en conjunto de estos sensores. Sin embargo, los intentos de lograr los mismos resultados con MAVs no han tenido tanto éxito debido a varias razones: la precisión y alta deriva de los Sistemas de Navegación Inercial (INS) en comparación con los basados en encoder de estimación, la carga de peso útil es limitada para la detección, la computación, y las dinámicas rápidas e inestables de vehículos aéreos, son los principales retos que deben abordarse.

Para muchas tareas de navegación es esencial la estimación de posición (*pose*), de la misma manera, la localización, el mapeado y el control son igualmente importantes. La técnica utilizada depende principalmente de los sensores disponibles en el MAV, que en la navegación aérea deben ser cuidadosamente elegidos debido a las limitaciones de carga útil y rangos de medida de los sensores. Debido al bajo peso y el consumo mínimo, la mayoría de los MAVs comerciales incorporan al menos una cámara monocular, por lo que, a parte de las técnicas SLAM, también existen otras técnicas de reconstrucción de mapas basadas en visión denominadas VSLAM que han ido ganando terreno en estos proyectos y son ampliamente utilizadas. Sin embargo, la mayoría de estos trabajos se han limitado a pequeñas áreas las cuales faciliten la obtención del VSLAM y entornos con suficiente luz solar para la cámara. Además, el tiempo de cálculo es demasiado alto para las dinámicas rápidas de los vehículos aéreos, lo que hace difícil controlarlos.

Aunque se han aplicado diferentes técnicas de SLAM, este proyecto se centra en la fusión de láser e IMU (Unidad de Medición Inercial) ([14], [16] y [17]) para estimar con precisión la posición de un MAV utilizando SLAM mediante el algoritmo Hector-Mapping. Para hacer frente a los requerimientos computacionales, el sistema se compone de una unidad de vuelo y un pequeño ordenador, por lo que el código se puede distribuir en diferentes nodos utilizando ROS (Robot Operating System). Con el fin de calcular la *pose*, el mapa en 2D se genera en base a láser. La posición estimada del MAV se filtra luego con la IMU e información obtenida a partir del láser utilizando un EKF (Extended Kalman Filter) para obtener una estimación de la *pose* 3D. En entornos que carecen de objetos característicos, como por ejemplo muebles, estanterías, etc, se ha demostrado que el EKF es robusto en este tipo de entornos y se obtienen buenos resultados.

1.1. Objetivos del proyecto

El objetivo general de este proyecto es realizar un sistema de localización y mapeado simultáneo para un robot aéreo (en este caso el AR Drone) en entornos interiores. Para ello se utilizará un sensor láser incorporado al robot además de los sensores embarcados en el mismo.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- **Entorno ROS:** Conocimiento del entorno de desarrollo robótico ROS.
- **Simulador:** Creación de un entorno de simulación para poder probar primero lo que se quiere conseguir a partir de un entorno simulado creado en 3D. Una vez conseguido la simulación, el siguiente paso será el acondicionamiento del robot real con los diferentes sensores para poder obtener el mismo resultado con los dispositivos reales.
- **Interfaz con el robot real:** Control básico del robot real mediante el envío de comandos

de velocidad para poder recorrer el entorno y obtener la lectura de los datos de los sensores del robot. Para poder procesar los datos del robot en el ordenador, se enviarán éstos mediante un pequeño ordenador que será implementado en el robot, en este caso se utilizará una Raspberry Pi que se conectará mediante red Wi-Fi del drone (red que el propio drone proporciona para su control) y que gracias a ella se podrán enviar los datos al ordenador de control.

- **Implementación del algoritmo Hector-Mapping:** Para la obtención de mapas en 2D se usará un algoritmo específico para ello, en este caso se utilizará el algoritmo "*Hector-Mapping*" el cual mediante los datos recogidos por el sensor láser Hokuyo URG-04LX permitirá obtener un mapa del entorno.

- **Estimador de posición 3D:** Una caraterística importante del SLAM es la localización simultánea del robot en el entorno creado, para ello se desarrollará un **estimador de posición 3D** basado en un "*Filtro de Kalman Extendido*".

- **Pruebas finales:** Una vez estén todos los dispositivos funcionando, se llevarán a cabo las diferentes pruebas que se han realizado previamente en simulación y se realizarán las diferentes comparativas de los resultados que ambos sistemas proporcionan.

1.2. Estructura del documento

El presente documento está divido en varias secciones, de las cuales la presente introducción es la primera, en la que se presenta el contexto, motivación y los objetivos de este proyecto.

En el capítulo 2 contendrá una breve reseña del estado del arte en la localización y el mapeado (SLAM). Se hablará en profundidad del entorno de desarrollo robótico ROS, los diferentes drones más conococidos en la actualidad y sus diferentes características. También una introducción teórica al Filtro de Kalman y Filtro de Kalman Extendido utilizado para la estimación de posición y una aclaración de los distintos mapas de navegación que nos encontramos para las aplicaciones del SLAM.

El capítulo 3 describe el algoritmo SLAM utilizado, que se trata del algoritmo Hector-Mapping. En este capítulo se hablará de las diferentes características que tiene el algoritmo y se explicará su uso en este proyecto.

El desarrollo propiamente dicho del proyecto se explicará en el capítulo 4. En este capítulo se hablará en profundidad de la plataforma robótica elegida y los diferentes sensores que se acoplarán a ella. También se describen los pasos necesarios para implementar los diferentes modelos 3D de la plataforma y los sensores al programa de simulación. Así, una vez explicada la simulación se explicará el montaje que ha sido necesario para el robot real y se hablará de la implementación del algoritmo en la plataforma para obtener los mismos resultados que en simulación. Por último, se describe el estimador de posición utilizado y las diferentes pruebas realizadas con él.

Las conclusiones del proyecto están recogidas en el capítulo 5, así como algunos trabajos futuros que se pueden realizar a partir de este proyecto.

El manual de usuario explicará los pasos necesarios para que se pueda reproducir este proyecto por cualquier lector y pueda entender mejor el proyecto.

En la sección de planos se introducirá los planos de los sensores y plataforma robótica utilizada, así como los algoritmos más significativos.

El pliego de condiciones contendrá los requisitos mínimos necesarios tanto de hardware como de software para la realización del proyecto.

Por último, se incluye un apartado para el presupuesto del proyecto que estimará el precio utilizado en el mismo.

Capítulo 2

Estado del Arte

En este capítulo se hablará de algunos MAV's que son usados en la actualidad para investigación y los principales métodos SLAM disponibles y, así mismo de diversas herramientas que son utilizadas a la hora de implementar estas técnicas.

2.1. MAV's comerciales para investigación

Este apartado describirá las diferentes plataformas de desarrollo que se conocen hoy en día y se centrará en concreto en las que se usará para llevar a cabo este proyecto.

Hoy en día el uso de estos vehículos no tripulados radio controlados está en auge, y pueden utilizarse tanto en actividades recreativas o incluso en trabajos de investigación que necesiten mayor conocimiento científico, introduciendo mejoras en hardware o software según sus necesidades. Diversas marcas trabajan en ello tales como "DJI", creador del Phantom, "Asc-Technology", que fabricó el cuadricóptero Pelican entre otros y "Parrot", diseñador del cuadricóptero AR Drone 2.0 y del Bebop Drone, entre otros.

A continuación se describirán brevemente las diferentes características que presentan cada uno de ellos, y se justificará el uso del AR Drone 2.0 de Parrot para la realización de este proyecto.

2.1.1. Phantom

Es un pequeño cuadricóptero fabricado por la empresa DJI[1] y mostrado en la figura 2.1.

A diferencia de otros cuadricópteros, Phantom no cuenta con cámara incorporada dando la ventaja al usuario de introducir la cámara que más se ajuste a sus necesidades mediante el uso de una plataforma destinada a ello, siendo Go-Pro la más idónea para ello aunque se puede introducir otro tipo de cámara que se ajuste a las características del Phantom.

[1] http://www.dji.com

Figura 2.1: Phantom Drone.

Se trata de un dispositivo controlado por radiocontrol que incorpora un GPS y una pequeña plataforma denominada Naza-M que cuenta con un microcontrolador y todos los sensores necesarios para un drone, como un barómetro, para calcular su altitud, un giróscopo, para su orientación y acelerómetros necesarios para el cálculo de velocidades. Todo ello forma en conjunto la IMU del Phantom.

Algunas características del Phantom a tener en cuenta son; la velocidad máxima de vuelo de 10 m/s y de despegue de 6 m/s, el peso máximo que puede soportar es inferior a 1,2 Kg y tiene una autonomía de 10 a 15 minutos dependiendo del peso que soporte.

Debido a la finalidad recreativa del Phantom, y a que en ROS no existen paquetes para poder realizar una investigación o proyecto con él, se descarta su uso en este proyecto.

2.1.2. Pelican

Pelican (figura 2.2) ha sido diseñado por la empresa Ascending Technologies[2] (AscTec) para combinar una gran potencia y espacio en un peso muy reducido debido a su cuerpo en fibra de carbono que lo hace más manejable y robusto. Puede alcanzar una velocidad de vuelo de 16 m/s y soporta una carga máxima de 1,65 Kg con una autonomía aproximada de 16 minutos. Tiene una estructura modificable gracias a su forma de torre que le permite introducir diversas plataformas en las cuales se pueden cargar los diversos sensores como cámaras, sensores láseres, GPS, batería, etc.

Figura 2.2: Pelican Drone.

[2]http://www.asctec.de/en/

Este cuadricóptero creado por AscTec está presente en muchos proyectos SLAM, reconstrucción 3D de entornos, vuelos autónomos, etc. Por ello, es interesante plantearse trabajar con él en este proyecto debido a todas las ventajas que proporciona en comparación con otros. Sin embargo, aunque Pelican se trate de un drone con unas características potentes su precio es elevado por lo que se ha decidido optar por elegir otra plataforma robótica.

2.1.3. Bebop Drone

Beebop Drone[3] (figura 2.3) es un pequeño cuadricóptero diseñado por Parrot, que puede alcanzar una velocidad máxima de 13 m/s, una autonomía de 22 minutos y con un peso de 420 gramos. Debido a su estabilidad y que cuenta con diversos sensores, entre ellos una cámara frontal de 14 mega píxeles y unos motores mejorados a su antecesor (AR Drone 2.0), este cuadricóptero fue una opción bastante factible a tener en cuenta a la hora de realizar este proyecto. Posteriormente fue descartado ya que el Bebop es relativamente nuevo, lo que limita el trabajo de investigación en ROS debido a que hay pocos trabajos realizados en este entorno, no existen drivers para control del Bebop mediante ROS y la SDK[4] aún está en desarrollo.

No obstante, no se ha descartado su uso para futuros proyectos en colaboración con el AR Drone 2.0 o que en un futuro sustituya a este para realizar el trabajo de SLAM con diversos sensores acoplados a él.

Figura 2.3: Bebop Drone.

2.1.4. AR Drone 2.0

Para finalizar esta sección se hablará del cuadricóptero AR Drone 2.0[5] (figura 2.4) que ha sido el elegido para la realización de este proyecto basándose en las distintas características presentadas y los diversos proyectos ya creados con él en el entorno ROS.

Fabricado por la empresa Parrot en 2010 cuenta con cuatro motores en configuración cuadricóptero "X" diferenciándose de otros debido a que está formado por una estructura modular que hace fácil su manejo e intercambio de piezas. Lleva una cámara frontal de 720p y una inferior que usa para su estabilización mediante el trabajo conjunto de los diferentes sensores implementados a través de un microprocesador internamente incorporado.

[3] http://www.parrot.com/es/productos/bebop-drone/
[4] https://github.com/Parrot-Developers/ARSDKBuildUtils
[5] http://ardrone2.parrot.com/

Figura 2.4: AR Drone 2.0.

Gracias a su aplicación "AR. FreeFlight", creada para sistemas operativos Android y iOS, permite al usuario tener un control total del cuadricóptero desde cualquier dispositivo móvil o Tablet, siendo este cuadricóptero utilizado mayoritariamente a nivel recreativo.

El AR Drone 2.0 puede alcanzar 11.11 m/s y pesa 420 gramos con la carcasa de protección interna. Su fácil manejo y programación, ya que cuenta con drivers específicos para el AR Drone 2.0 en el entorno robótico elegido para el proyecto (ROS), han sido las razones por las que se ha elegido esta plataforma robótica para este proyecto.

Posteriormente, en la sección 4.1.1, se describen más detalladamente las especificaciones y características del AR Drone 2.0.

2.2. Entornos de desarrollo para aplicaciones robóticas

El objetivo principal de cualquier aplicación robótica es proporcionar una serie de conocimientos o autonomía a un robot. Actualmente es posible encontrar soluciones comerciales o en código abierto que permiten realizar esta tarea más fácilmente mediante distintas librerías o módulos de trabajo que estos códigos proporcionan, provocando que la programación de bajo nivel, que anteriormente se realizaba para obtener el mismo resultado, pasase a un segundo plano.

Con el objetivo de realizar este trabajo de forma más sencilla y reducir tanto los tiempos de desarrollo como obtener soluciones a problemas futuros a partir de diversas simulaciones previas, la comunidad robótica comenzó a desarrollar plataformas de desarrollo que proporcionasen un espacio de simulación mediante modelos informáticos de implementaciones reales de diversos sensores y modelos robóticos, con las mismas características que los reales.

Algunas plataformas robóticas han sido desarrolladas para solucionar estos problemas como es el caso de Player[6] que se trata de un framework de programación de código abierto y libre, para el desarrollo de software enfocado a plataformas robóticas, el cual tiene dos simuladores asociados: Stage y Gazebo[7]. Player es un servidor de red para el control de un robot que proporciona una potente interfaz con una gran variedad sensores y actuadores, como por ejemplo robots. Debido a que está basado en socket TCP (Protocolo de Control de Transmisión), Player utiliza un modelo de cliente/servidor para comunicarse con los robots mediante programas de control que pueden ser escritos en varios lenguajes de programación y ser ejecutados en cualquier ordenador con conexión a la red del robot. Además Player soporta conexión simultánea de distintos clientes a dispositivos haciendo posible la comunicación colaborativa entre éstos.

Como se ha comentado anteriormente, Player cuenta con dos simuladores para poder realizar

[6] http://playerstage.sourceforge.net
[7] http://gazebosim.org/

este desarrollo robótico de forma más real como Stage y Gazebo, simuladores que funcionan en espacios de trabajo 2D y 3D respectivamente. Stage es un simulador robótico que proporciona un mundo virtual de una población de robots móviles y sensores incluyendo diversos objetos para la detección y manipulación de los robots. Gazebo cuenta con un mundo tridimensional que permite introducir y simular modelos robóticos en 3D y diversos tipos de sensores haciendo posible un fácil diseño y uso del robot en algunos de los escenarios que el simulador proporciona en su base de datos.

Además de Player, existen otros softwares de código abierto, como es el caso de CARMEN[8], que controla robots móviles basándose en un software modular que permite realizar diferentes aplicaciones cómo la navegación, mapeado, localización, etc. Otros entornos robóticos son Microsoft Robotic Studio, que ofrece servicios de simulación 3D, y ORCA[9], que se trata de un entorno en código abierto.

ROS ha evolucionado a partir de entornos de desarrollo como Player y CARMEN, convirtiéndose así en una de las plataformas robóticas más utilizada hoy en día y por ello se ha decidido utilizar este entorno para la realización de este proyecto.

2.2.1. ROS

ROS, siglas en inglés de "*Robot Operating System*", nace en 2007 como proyecto robótico en la Universidad de Stanford bajo el nombre de *Switchyard*. A finales de 2007 el proyecto se desarrolló principalmente en el instituto de investigación Willow Garage[10] y a partir de entonces se mantiene por Open Source Robotics Foundation bajo licencia BSD.

Se trata de un framework, lo que se conoce también como plataforma software, que proporciona al usuario una serie de herramientas de visualización, librerías, monitorización y análisis, todas ellas en código abierto específicas para desarrollo de software robótico.

La estructura de ROS es modular, se crean proyectos en paquetes los cuales contienen los códigos necesarios para desarrollar aplicaciones robóticas según las necesidades de cada usuario. Debido a su estructura de software abierto, ROS permite utilizar diversos paquetes de distintos proyectos de otras plataformas de software abierto como Gazebo, OpenCV[11], Player/Stage, entre otros, para poder trabajar en conjunto y así realizar proyectos de cualquier grado de dificultad.

Trabaja como si se tratase de un sistema operativo ya que proporciona servicios estándares tales como abstracción del hardware, comunicación a través de mensajes entre procesos, mantenimiento de paquetes, control de dispositivos de bajo nivel e implementación de funcionalidad que son propios de éstos. Se habla de que trabaja como un sistema operativo pero en realidad no lo es ya que se instala sobre otro, generalmente Linux, y por preferencia en Ubuntu[12], y por ello también recibe el nombre de Meta-Sistema Operativo, como muestra la imagen 2.5.

[8] http://carmen.sourceforge.net/
[9] http://orca-robotics.sourceforge.net/
[10] https://www.willowgarage.com/
[11] http://opencv.org/
[12] http://www.ubuntu.com/

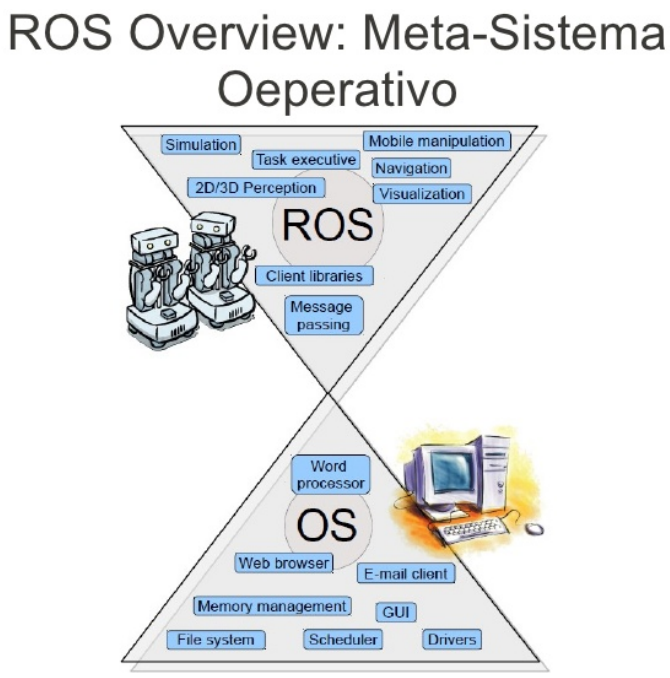


Figura 2.5: ROS Meta-Sistema Operativo.

Tras hacer una breve descripción de ROS es útil realizar una comparativa de ventajas y desventajas que ROS proporciona para hacer una justificación de por qué se ha elegido como entorno de desarrollo para realizar este trabajo.

Las ventajas más características de ROS lo definen como un proyecto de código abierto que permite a todo el mundo acceder a él sin necesidad de ningún coste, permite programar diversos drivers, abordar problemas de alto nivel, abstracción del hardware reduciendo el tiempo de infraestructura, centrándose más en la investigación y permite la comunicación entre distintos robots que son controlados por ordenador.

Por el contrario, ROS no cuenta con simuladores específicos provocando que se usen simuladores externos, en ocasiones la documentación es incompleta ya que se trata de un proyecto en continuo desarrollo y algunos drivers de dispositivos no existen, haciendo complicado el trabajo con algunos sensores.

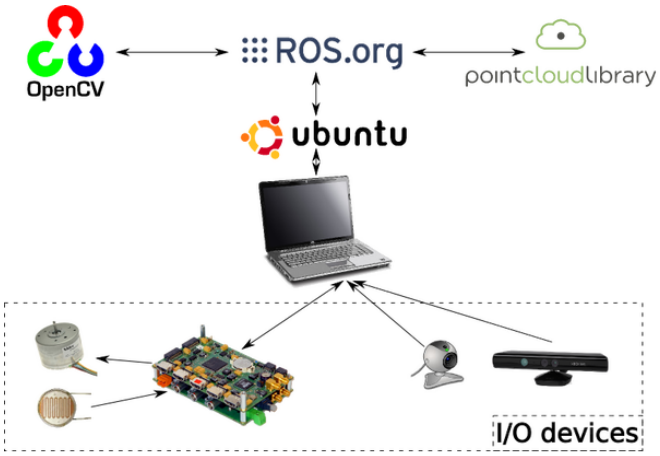


Figura 2.6: Comunicación con Dispositivos

ROS es una red de procesos que se acoplan libremente utilizando estructura *peer-to-peer*, también conocida como estructura P2P la cual consiste en una comunicación mediante "*nodos*" que se comportan iguales entre sí, funcionando sin clientes ni servidores fijos. Implementa diferentes tipos de comunicación: de forma síncrona se comunica mediante "*servicios*", la transmisión asíncrona la realiza mediante "*topics*" y el almacenamiento de datos en un "*servidor de parámetros*" mediante un nodo "*maestro*" que controla a todos los servicios, mensajes, nodos y topics.

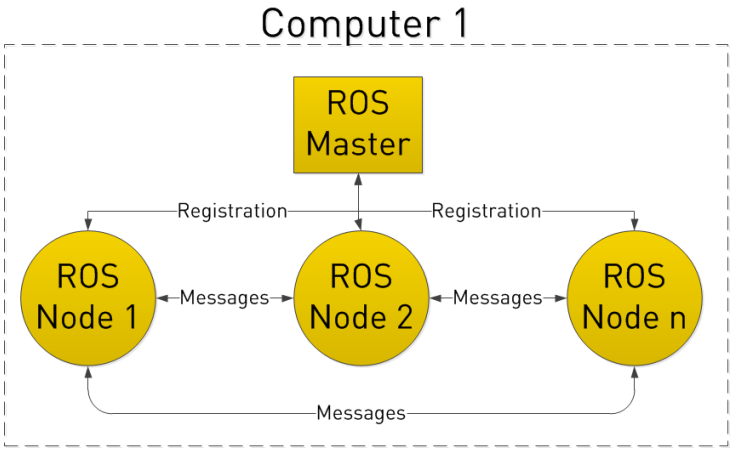


Figura 2.7: Comunicacion en ROS

A continuación se describirán brevemente los conceptos básicos para entender ROS.

2.2.1.1. Conceptos de ROS

Para poder entender mejor ROS hay que tener en cuenta que tiene tres niveles conceptuales:

- Sistema de Archivos (*Filesystem Level*). Organización de los diferentes archivos en el sistema.
- Nivel de Computación Gráfica (*Computational Graph Level*). Componentes de la red.
- Nivel Comunitario (*Comumunity Level*). Los usuarios comparten software mundialmente.

Filesystem Level:

Este concepto se refiere a la organización de los archivos dentro del sistema y principalmente se organizan en paquetes y pilas:

- **Paquetes (packages):** Los paquetes son la unidad más "atómica" y principal de construcción del sistema de archivos de ROS. Un paquete puede contener procesos (nodos),una biblioteca dependiente del sistema ROS, bases de datos, archivos de configuración, o cualquier otra cosa que se organice de manera útil en conjunto.
- **Pilas (stacks):** Conjunto de paquetes que tienen como objetivo la organización de un proyecto (o varios proyectos) para llevar a cabo funciones de alto nivel.

Otro archivo importante en ROS es el *manifest*, el cual contiene información general sobre el contenido de un paquete o un repositorio. Este archivo puede contener una breve descripción de la utilización del paquete, su tipo de licencia y las dependencias con otros paquetes. Esta información sobre las dependencias son importantes porque generalmente los paquetes dependen de otros paquetes y éstos deben estar instalados para funcionar correctamente.

Computational Graph Level:

El nivel de Computación Gráfica es la red de procesos de ROS la cual se ejecuta mediante *peer-to-peer* (P2P). Los conceptos básicos para entender este nivel son los siguientes:

- **Nodos (node):** Los nodos son los encargados de realizar la computación propiamente dicha. ROS esta estructurado modularmente de forma que un robot puede comprender muchos nodos que controlen distintas partes del robot, por ejemplo, un nodo específico para el movimiento del robot, otro para las distintas articulaciones del mismo y así sucesivamente. Un nodo de ROS está escrito usando una biblioteca específica de cliente, tales como *roscpp* (C++) o *Rospy* (Python).
- **Maestro (master):** Es el nodo principal. El master proporciona el registro de nombres y de consulta para la computación gráfica. Sin éste, los nodos son incapaces de trabajar ya que no encontrarían los *mensajes* necesarios o los *servicios* que una aplicación los requiera. El master, en general, se ejecuta mediante el terminal[13] con el comando **roscore** aunque hay excepciones en las que se ejecuta solo mediante la carga de archivos **.launch**.
- **Servidor de parámetros (parameter server):** Permite el almacenamiento de datos en una localización central.
- **Mensajes (messages):** Comunicación entre nodos. Se trata de una estructura de datos la cual puede tener tipos numéricos y estructuras anidadas arbitrariamente.
- **Temas (topics):** Los nodos comparten información mandando y recibiendo mensajes. Estos mensajes son publicados con un nombre especial llamado Topics. Por lo tanto, los nodos pueden publicar o suscribirse a un topic específico para obtener los mensajes deseados. Incluso, un mismo nodo puede estar suscrito a distintos topics.
- **Servicios (services):** Los servicios proporcionan una comunicación mediante un método de pregunta/respuesta. Es decir, está formado por dos partes: un mensaje que es el que pregunta y otro mensaje que es el que responde, o sea, cuando un nodo quiere llamar a un servicio, tiene que llamarle y esperar la respuesta. A diferencia de la comunicación de los topics, un nodo sólo puede llamar a un servicio bajo un nombre particular.
- **Bolsas (bags):** Se trata de un formato especial que permite al usuario la opción de guardar o reproducir mensajes de datos de ROS. Por ello, los *bags* tienen una gran importancia a la hora de hacer pruebas para desarrollo y algoritmos.

La imagen 2.8 muestra la comunicación entre 2 nodos, haciendo referencia a lo anteriormente comentado. El nodo de la izquierda pubica mensajes en un topic, mientras que el nodo de la derecha se subsrcibe a este topic para obtener los mensajes. Por otro lado, el nodo de la derecha se comunica mediante servicios con el nodo de la izquierda invocando un mensaje y obteniendo una respuesta de este último.

[13]Modo de acceso al sistema Linux sin necesidad de interfaz gráfica.

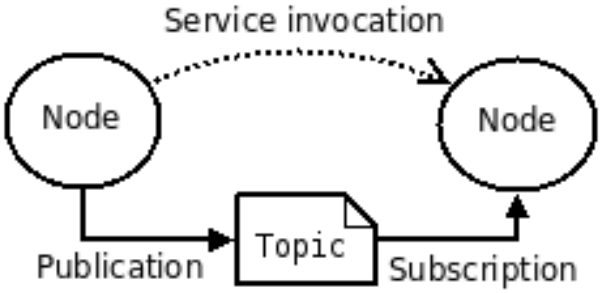


Figura 2.8: Comunicacion entre nodos.

Comumunity Level:

Una característica de ROS es la creación de una comunidad robótica mundial la cual comparte software y códigos para la creación de diferentes paquetes.

Existen varias formas de obtener los recursos de ROS provinientes de esta comunidad:

- **Distribuciones:** Permite obtener una serie de pilas que contienen diferentes paquetes que se pueden instalar en el sistema.
- **Repositorios:** La existencia de diferentes repositorios en internet facilita la comunicación de paquetes en esta comunidad pudiendo obtener directamente los paquetes o modificarlos. Por ejemplo *github*[14], la cual proporciona diversos paquetes en una red social. A la hora de ser descargados e instalados, las pilas se agrupan en repositorios, que son equivalentes a los repositorios de Linux. De hecho los repositorios de ROS pueden ser descargados equivalentemente a los de Linux, directamente por línea de comandos o mediante el administrador de descargas.
- **ROS Wiki:** Comunidad de ROS que contiene la información principal de ROS en la que se puede encontrar diferentes tutoriales y paquetes.
- **ROS Answer:** Se trata de una red de comunicación entre usuarios de ROS en la que se puede realizar preguntas acerca de proyectos basados en ROS.

2.2.1.2. Herramientas de ROS

ROS proporciona herramientas para facilitar la implementación del sistema robótico. Estas herramientas permiten a los usuarios simular, lanzar varios nodos a la vez y visualizar el flujo entre los nodos. Algunas de las herramientas que posee se presentan a continuación.

- **roscore:** Herramienta que ejecuta el núcleo de ROS. Este comando es muy importante y permite inicializar la comunicación entre nodos, servicios, parámetros, etc, mediante el nodo “maestro”.

```
$ roscore
```

- **roscreate-pkg:** Mediante esta herramienta el usuario puede comenzar la creación de los paquetes que necesite, pudiendo además especificar las dependencias de otros paquetes.

[14]https://github.com/

```
$ roscreate-pkg [nombre del paquete] [dependencias]
```

- **rospack:** Se puede obtener información de un paquete mediante esta herramienta.

```
$ rospack [comando] [paquete]
```

- **rosstack:** Información de la "pila" que se desee.

```
$ rosstack [comando] [stack]
```

- **roscd:** Permite acceder a un paquete.

```
$ roscd [paquete]
```

- **rosls:** Herramienta que permite al usuario ver el contenido de un paquete.

```
$ rosls [paquete]
```

- **rosnode:** Muestra una lista los nodos en ejecución, información de un nodo o permite eliminar nodos de la ejecución.

```
$ rosnode [comando] [nodo]
```

- **rosdep:** Permite descargar dependencias de paquetes.

```
$ rosdep install [paquete]
```

- **rostopic:** Herramienta útil para controlar "topics". Se puede obtener una lista de todos los "topis" en ejecución, así cómo publicar mensajes en ellos, obtener información o leer los datos de dichos "topics".

```
$ rostopic [comando] [topic] [argumentos]
```

- **rosservice:** Al igual que los "topics", se puede obtener una lista de los distintos servicios que se están ejecutando, información y utilizar dichos servicios para un fin.

```
$ rosservice [comando] [servicio] [argumentos]
```

- **rosmsg:** Permite obtener la información de un tipo de mensaje.

```
$ rosmsg [comando] [tipo de mensaje]
```

- **rosrun:** Permite ejecutar un archivo ejecutable de un paquete específico.

```
$ rosrun [paquete] [ejecutable]
```

- **roslaunch:** ROS permite ejecutar varios nodos a la vez. Mediante esta herramienta se puede ejecutar archivos con formato *.launch* en estos archivos se puede configurar varios nodos para poder ejecutarlos a la vez facilitando al usuario el uso del sistema global de ROS.

```
$ roslaunch [paquete] [archivo .launch]
```

2.2.1.3. Versiones de ROS

Al tratarse de un proyecto en continuo desarrollo existen diferentes versiones de ROS que se han ido actualizando a lo largo de los años.

En la tabla 2.1 se puede ver una evolución de las distintas versiones de ROS.

Versiones	Año
Box Turtle	Marzo, 2010
C Turtle	Agosto, 2010
Diamondback	Marzo, 2011
Electric	Agosto, 2011
Fuerte	Abril, 2012
Groovy	Diciembre, 2012
Hydro	Septiembre, 2013
Indigo	Julio, 2014
Jade	Mayo, 2015

Tabla 2.1: Versiones de ROS.

En este proyecto se ha decidido utilizar Hydro Medusa debido a que al comenzar dicho proyecto era la versión más reciente hasta entonces, pudiendo utilizar la mayoría de los paquetes sin necesidad de instalar dependencias externas. Sin embargo, la investigación ha de continuar y se han empezado a realizar pruebas en la versión siguiente (Indigo) ya que con el paso del tiempo se ha ido consolidando hasta alcanzar el nivel de Hydro en cuanto a dependencias de paquetes.

2.2.1.4. Gazebo

Gazebo es un entorno de simulación para aplicaciones robóticas. Permite una rápida ejecución de algoritmos, diseño de modelos robóticos y pruebas de regresión[15] usando escenarios realistas. Además, Gazebo ofrece la posibilidad de simular con precisión y eficiencia poblaciones de robots en entornos complejos tanto en interiores como en exteriores.

Figura 2.9: Icono de Gazebo.

Se trata de un entorno de simulación con un motor gráfico de alta calidad en el que se puede simular "físicas", así pues, Gazebo permite simular fuerzas gravitatorias, aceleraciones,

[15]Realización de diferentes pruebas de software que tiene el objetivo de descubrir errores.

velocidades, etc. Se puede apreciar en la figura 2.10 cómo trabaja con escenarios realistas, como una gasolinera en la que vuela un MAV.

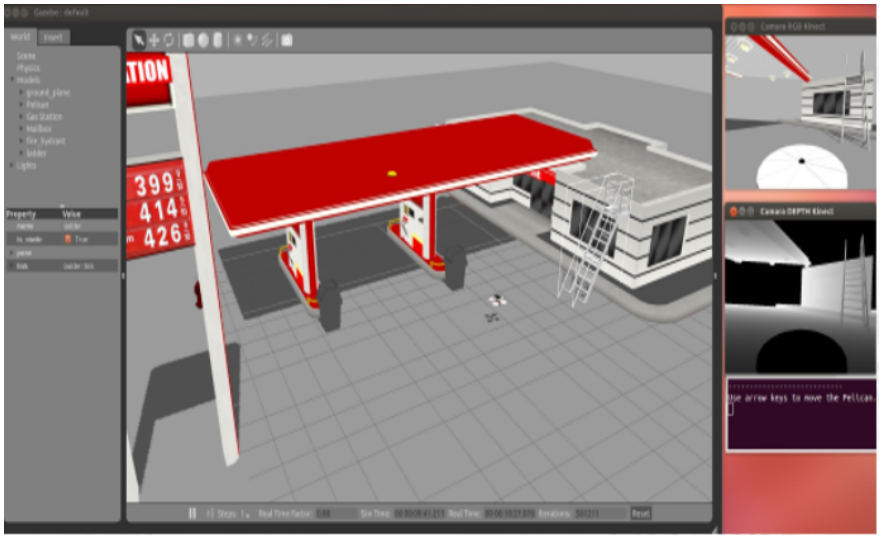


Figura 2.10: Entorno de simulación de Gazebo simulando una gasolinera y el vuelo de un cuadricóptero. Se pueden ver además las imágenes que la cámara del cuadricóptero capta, en este caso se usa una Kinect acoplada al modelo del cuadricóptero.

Gazebo está implementado en ROS mediante la utilización de paquetes. Según la versión que se esté utilizando de ROS, existen distintos paquetes para poder instalar Gazebo. En la tabla 2.2 se explica qué paquete se ha de instalar según la versión.

Versiones de ROS	Paquetes en ROS
Electric Fuerte Groovy	gazebo
Hydro Indigo Jade	gazebo_ros_pkgs

Tabla 2.2: Diferentes paquetes de Gazebo en ROS según versiones.

Como se puede apreciar, las primeras 3 versiones de ROS trabajan con el paquete *gazebo*[16] y las 3 últimas, como es el caso de este proyecto, se usa el paquete *gazebo_ros_pkgs*[17].

Otro punto importante de Gazebo es la manera de trabajar con los modelos en 3D de los robots y los distintos escenarios que se crean. Utiliza un formato especial conocido como *Unified Robot Description Format (URDF)* para poder simular los modelos de los diferentes robots. Este formato tiene una estructura de programación que establece las diferentes articulaciones del robot y sus partes.

Para poder utilizar Gazebo, desde un terminal de Ubuntu podemos ejecutar el siguiente comando que nos abrirá un espacio de trabajo vacío en Gazebo, tal y como se puede observar en la figura 2.11.

[16] http://wiki.ros.org/gazebo
[17] http://wiki.ros.org/gazebo_ros_pkgs

```
$ rosrun gazebo gazebo
```

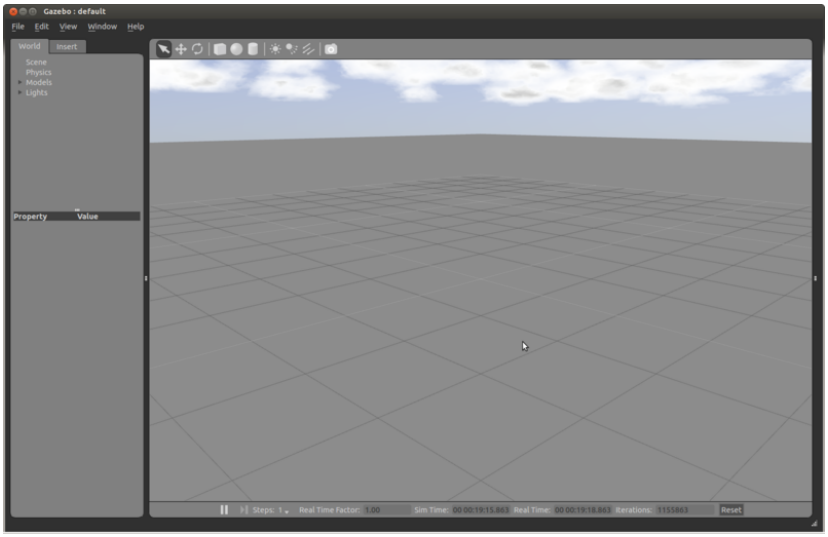


Figura 2.11: Espacio de trabajo vacío de Gazebo.

Esta interfaz es muy intuitiva y permite al usuario introducir una gran cantidad de modelos creados desde el servidor propio de Gazebo. Este servidor contiene una base de datos en la que se pueden encontrar, por ejemplo, modelos 3D de oficinas de *Willow Garage*, modelos de una casa, robots como el P2DX de Pioneer, etc, sin necesidad de crear estos modelos con otro programa. Sin embargo, estos modelos no son controlables y sólo son utilizables como parte del entorno, es decir, si se introduce un modelo de un robot éste no cuenta con los topics necesarios para su manejo y tan solo es el modelo 3D.

Figura 2.12: Base de datos de modelos de Gazebo.

Debido a que Gazebo presenta unas características muy "potentes", tanto en motores gráficos como simulaciones muy realistas, y el fácil uso de su interfaz, este entorno de simulación 3D ha sido el elegido para la realización de este proyecto.

2.2.1.5. Rviz

Es un visualizador 3D que permite visualizar datos de diferentes sensores y los distintos estados de información de ROS. Mediante Rviz[18] se puede visualizar modelos virtuales robóticos y mostrar "on-line" los datos recibidos por los diferentes sensores de robots reales, como la visualización de imagenes obtenidas a partir de una cámara, nube de puntos de sensores láseres, mapas, o las diferentes "transformaciones" que forman el modelo virtual del robot, entre otros.

Figura 2.13: Icono de Rviz.

Rviz proporciona una interfaz sencilla para elegir la información que se quiere visualizar.

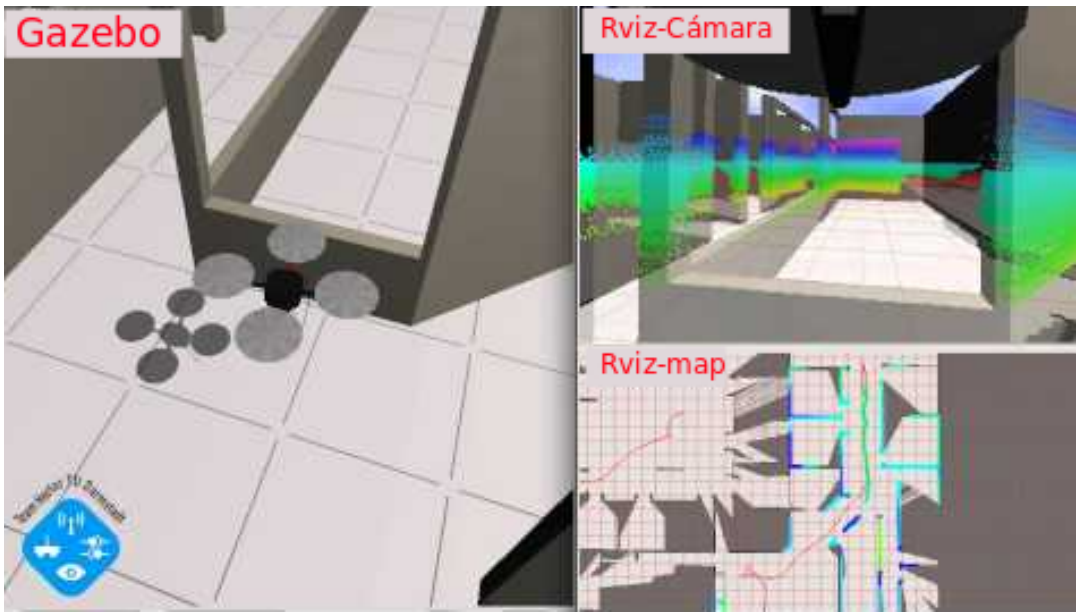


Figura 2.14: Visualización en Rviz.

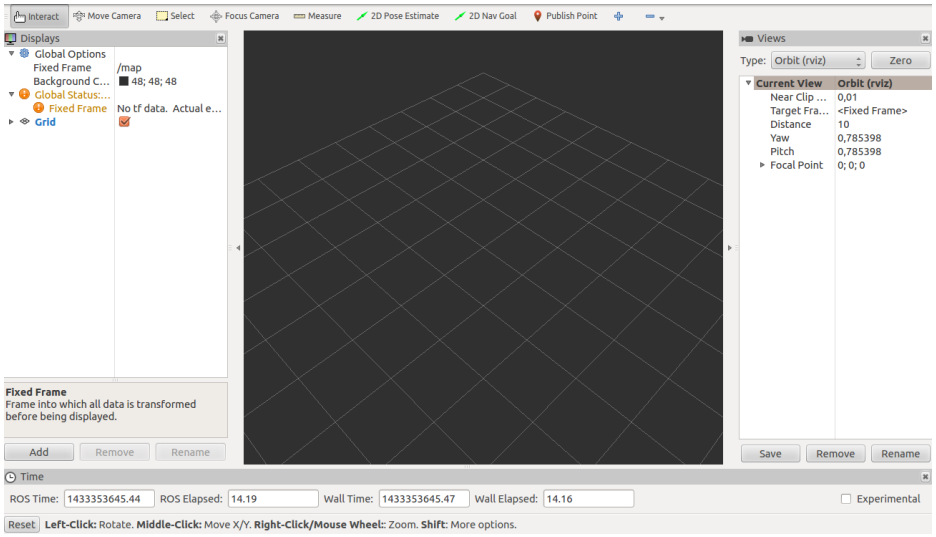
En la figura 2.14 se puede observar cómo se visualizan los diferentes datos obtenidos de los sensores del MAV, simulado en Gazebo, y cómo se reconstruye el mapa 2D a partir de ellos.

Para poder usar esta herramienta, que se encuentra en ROS, basta con lanzar un comando por el terminal:

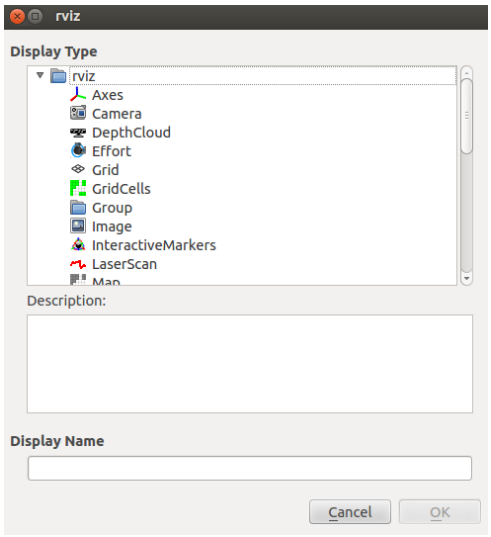
```
$ rosrun rviz rviz
```

[18] http://wiki.ros.org/rviz

Una vez ejecutado este comando, se abrirá el programa (ver figura 2.15.a) con un escenario vacío en el que podremos elegir los diferentes datos que queremos visualizar (ver figura 2.15.b).



(a) Rviz vacío.



(b) Diferentes datos de Rviz.

Figura 2.15: Rviz

2.3. Métodos de localización y mapeado

Los robots, en este caso un cuadricóptero, normalmente realizan trabajos de forma autónoma en entornos dinámicos y complejos. La localización y la reconstrucción simultánea de un mapa del entorno son muy importantes para realizar los movimientos autónomos necesarios.

Antes de que existiesen métodos para resolver estos problemas, los investigadores daban una serie de parámetros previos al robot en función del entorno de trabajo así como la previa construcción del mapa haciendo que fuese difícil implementar la perspectiva del

robot al entorno. Esto hacía carecer de sentido la autonomía del robot ya que todos los objetivos que el robot debería haber realizado (mapeado y localización) eran previamente construidos.

Por ello, se ha avanzado mucho en este ámbito de investigación para poder dar al robot cierto sentido y que sea él mismo el que genere el mapa y una localización mediante sensores destinados a ellos.

En general, un robot necesita dos tareas principales para realizar un movimiento autónomo:

- ***Mapeado:*** El *mapeado* es el problema de integrar la información obtenida a través de los sensores del robot. Los aspectos centrales del *mapeado* son la representación del entorno y la interpretación de los datos obtenidos por medio de los sensores, creando mediante algoritmos, un mapa del entorno.
- ***Localización:*** La *localización* es el problema de estimar la pose[19] del robot en un mapa. Dentro de la localización se puede diferenciar dos tipos de localización; *localización local*, que se refiere a la posición inicial conocida, y *localización global*, posición del robot desconocida en el entorno.

En la imagen 2.16 muestra cómo se integran los diferentes problemas para realizar el mapeado y la localización del robot en un entorno.

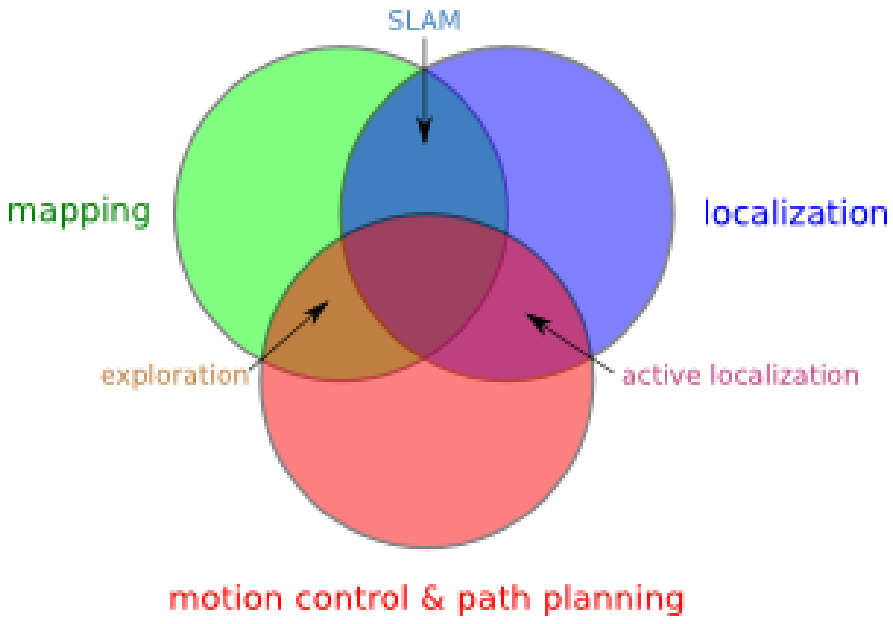


Figura 2.16: Implementación del problema del mapeado y localización.

En la figura 2.16 se explica el problema del SLAM. La zona central del diagrama representa los enfoques integrados que buscan soluciones simultáneas al problema de planificación, localización y mapeado.

A la hora de realizar un proyecto en el que los robots necesiten autonomía es necesario realizar lo anteriormente descrito. Por ello, es imprescindible establecer las bases del SLAM[20]. Se han propuesto un gran número de modelos, enfoques y técnicas que ayuden a solucionar los problemas del SLAM. A menudo, se usan soluciones probabilísticas al dar soluciones

[19] En robótica móvil: se refiere a la posición y orientación del robot en conjunto, éstos son importantes para determinar la localización del robot en el entorno.

[20] Siglas en inglés: Simultaneous Localization And Mapping

más fiables, reduciendo el error de una manera más eficaz. Los elementos fundamentales de estas soluciones probabilísticas son, la definición de un ***modelo de observación***, que proporcione las probabilidades de las diferentes lecturas de los sensores dada una posición en el entorno definido, y la definición de un ***modelo de movimiento***, que proporcione la probabilidad de la siguiente posición del robot, dada la posición actual y la acción que se ha ejecutado. Existen varios tipos de mapas que pueden generarse en función de los sensores utilizados.

A continuación se hace una breve explicación de los mapas más utilizados.

Mapas Topológicos: Representan las características más relevantes del entorno capturadas por el robot móvil usando un grafo. Los nodos representan lugares distintivos del entorno, y los arcos caminos que serán recorridos por el robot uniendo los distintos lugares. Estos lugares representativos del entorno son seleccionados en base a sus características distinguibles. Por ejemplo, la figura 2.17 muestra los lugares donde se han detectado *landmarks*[21] al realizar un experimento con un robot móvil.

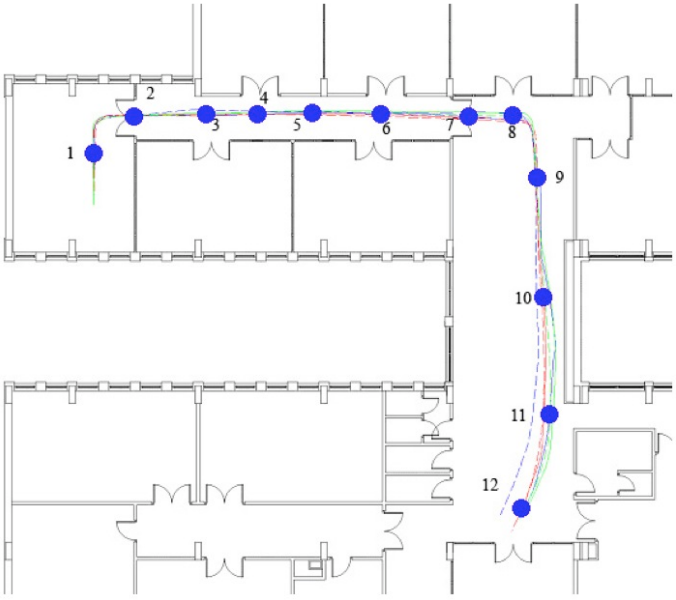


Figura 2.17: Ejemplo de un mapa topológico.

Estos tipos de mapas ofrecen información del entorno y dicha información es utilizada por el robot para conocer su localización. Así mismo, permiten planificar una trayectoria del robot proporcionando una interfaz natural para la interacción humana debido a que los *landmarks* registrados por el robot del entorno se almacenan permitiendo que se manden comandos, como por ejemplo *"ir a la posición X"* del mapa. Como desventaja o inconveniente es dificil representar determinados entornos debido a su complejidad en representar zonas con pocos elementos característicos, dependen en exceso de los *landmarks* obtenidos en el proceso de mapeo y son sensibles a la aparición de elementos no modelados, es decir, la aparición de obstaculos imprevistos suponen una información sensorial distinta a la modelada produciendo que el robot pierda su localización.

Mapa de Características o Mapas Geométricos: Mediante elementos altamente característicos el robot puede obtener una representación del mapa y su localización en él. La

[21] Conocido como punto de referencia en la representación de un mapa.

localización se lleva a cabo extrayendo las características de los datos observados asociándolas después a las características del mapa. La mayoría de los robots usan una cámara como sensor principal, aunque también, suelen usar sensores láser.

Estos tipos de mapas no permiten planificar trayectorias y no trabajan bien en entornos donde los objetos no se pueden aproximar a un modelo geométrico fácilmente descrito, como se puede ver en la figura 2.18. Sin embargo, el espacio libre entre objetos no es representado lo cual no añade una carga computacional al proceso de localización.

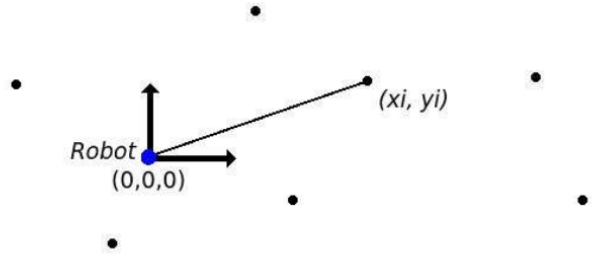


Figura 2.18: Ejemplo de un mapa de características.

Mapa de celdas de ocupación: Estos tipos de mapas se basan en dividir en unidades de tamaños predefinidos (se usan algoritmos destinados a ellos y son conocidos como *Occupancy Grid Mapping*) el espacio, siendo cada "celda" clasificada como ocupada o vacía con un determinado nivel de confianza o incertidumbre. La localización se hace comparando los datos de cada observación con el mapa, usando técnicas de correlación cruzada.

La imagen 2.19 es un ejemplo de mapa de celdas basado en el algoritmo ***Hector Mapping*** el cuál usa este tipo de mapas para representar entornos. La localización la consigue mediante el paquete de ROS, que se usará en este proyecto, conocido como ***Hector Slam*** usando los distintos sensores del robot para conseguir una localización fiable.

Figura 2.19: Ejemplo de mapa de celdas basado en Hector Mapping.

Los mapas de celdas pueden trabajar en entornos dinámicos y pequeños, se trata de algoritmos robustos y de implementación sencilla, se pueden planificar trayectorias al ser capaces

de distinguir zonas ocupadas o vacías y el resultado final es bueno. Como inconveniente, estos mapas no ofrecen un modelo exacto de incertidumbre por lo que en periodos muy largos tienden a diverger.

2.3.1. Simultaneous Localization and Mapping (SLAM)

El SLAM consiste en el proceso que realiza un robot para la reconstrucción de un mapa de un entorno, y al mismo tiempo, usar ese mapa para calcular su localización dentro de él, usando algoritmos destinados a ello, mediante el uso simultáneo de distintos sensores del robot y la confianza del usuario hacia esos sensores. Debido a esta complejidad, el SLAM se ha convertido en el campo de investigación más activo de las últimas dos décadas.

La última década ha experimentado un gran avance de los métodos SLAM, centrándose principalmente en mejorar la eficiencia computacional garantizando al mismo tiempo la estimación consistente y precisa para el mapa y el robot. Gracias a las investigaciones realizadas por Randall Smith, Peter Cheeseman [4] y Hugh Durrant-Whyte [5] se han establecido las bases estadísticas para describir relaciones espaciales entre los diferentes elementos del entorno y manipular las incertidumbres (o errores) asociados a sus respectivas posiciones.

Estos investigadores nombrados, han seguido trabajando en el ámbito del SLAM. Así pues, Hugh Durrant-Whyte y Tim Bailey en 2006 publicaron dos artículos [6] y [7] en los cuáles explican y resumen los conceptos básicos del SLAM y los diferentes métodos junto con las herramientas utilizadas para su implementación. Durrant-Whyte y Bailey explican el problema esencial del SLAM, como se puede ver en la figura 2.20.

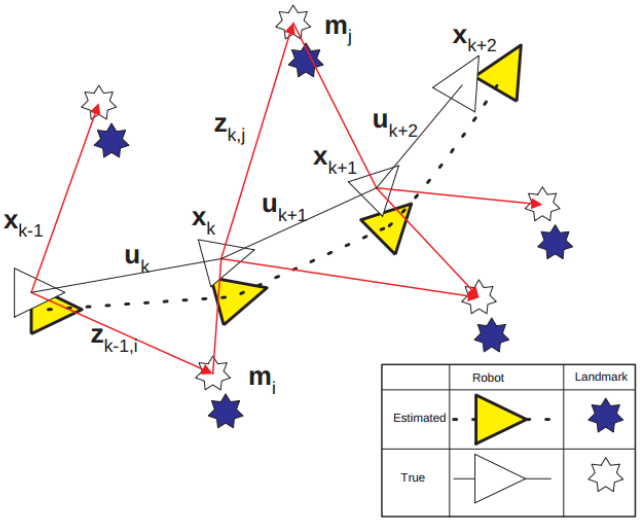


Figura 2.20: Problema esencial del SLAM.

La figura 2.20 explica el problema esencial del SLAM. Se requiere una estimación simultánea de las localizaciones del robot y *landmarks*. Las localizaciones verdaderas nunca se conocen o se miden directamente. Las observaciones se realizan entre las localizaciones verdaderas del robot y *landmarks*. Con ello se puede ir reduciendo el error de estimación de la posición del robot.

Como ya se ha comentado anteriormente en la sección 2.3, el **modelo de observación** proporciona las probabilidades de las diferentes lecturas de los sensores dada una posición

en el entorno, que se define con la siguiente ecuación:

$$P(z_k|x_k, m) \tag{2.1}$$

A su vez, el **modelo de movimiento**, el cuál proporciona una probabilidad de la siguiente posición del robot, se define con la siguiente ecuación:

$$P(x_k|x_{k-1}, u_k) \tag{2.2}$$

Donde **x** es la localización del robot, **u** las entradas de control y **z** el registro de todos los *landmarks* (o medidas).

Las soluciones dadas al problema probabilístico del SLAM implican una búsqueda adecuada a la ecuación 2.1 presentada por Durrant-Whyte y Bailey y al modelo del movimiento de la ecuación 2.2.

Las mejores soluciones que se han obtenido para el SLAM han sido con algoritmos basádos en ***Filtro de Bayes*** y los métodos probabilísticos de localización y generación de mapas introducidos por Thrun [8]. Éstos consiguen hacer frente a todas las fuentes de incertidumbre involucradas en el proceso.

2.3.1.1. Filtro de Bayes

Para modelar un entorno desconocido es importante saber que existe una incertidumbre debida a la imperfección de los sensores y modelos empleados. Si fuesen perfectos, se podría obtener una representación del entorno, y de los distintos objetos que le rodean, perfecta conociendo todas las posiciones de los elementos característicos de dicho entorno. El libro *Probabilistic Robotics* [9] se hace un repaso de las técnicas probabilísticas de reconstrucción de mapas.

En cuanto a la localización, si se conocen con exactitud las medidas de orientación, movimiento de ruedas o cuánto en realidad se ha movido el robot, permitiría obtener una localización exacta al mínimo movimiento del robot, pudiendo construir con todo esto un modelo "perfecto" del entorno acumulando sucesivas medidas.

Debido a las imperfecciones, que prácticamente son inevitables, se impone la necesidad de crear una relación de estas incertidumbres en las soluciones obtenidas. Por ello, se parte de la base de que la posición del robot y los elementos que modelan el entorno son variables aleatorias.

El principio básico que se encuentra en toda solución exitosa del SLAM es la ***"Regla de Bayes"*** la cual permite inferir el estado de un objeto x en base a una serie de observaciones z sobre él. La inferencia Bayesiana necesita que exista una relación entre x y z modelada como una función de densidad de probabilidad. Así pues, esta regla sigue la siguiente ecuación:

$$P(x|z) = \frac{P(z|x) * P(x)}{P(z)} \tag{2.3}$$

La anterior regla descrita, indica que se pueden resolver los diferentes problemas simplemente realizando la multiplicación de dos términos:

- Verosimilitud *P(x|z)*: expresa la probabilidad de obtener la medida z bajo una hipótesis expresada en el estado x.
- Grado de confianza que damos a que x sea precisamente el caso antes de recibir los datos *P(x)* a priori.

El problema de la representación de un entorno (mapa) está sujeto a una variable adicional que la ecuación 2.3 no contempla y se trata del tiempo. De modo que la generalización temporal del ***teorema de Bayes***, recibe el nombre de ***Filtro de Bayes***. Se trata de un estimador de posición recursivo, es decir, que calcula la secuencia de distribuciones de probabilidad a posteriori de magnitudes que no pueden ser directamente observadas, en función de otras que sí pueden serlo. La ecuación 2.4 es la ecuación genérica del *Filtro de Bayes* y calcula la probabilidad de un estado x en un instante t de tiempo.

$$P(x_t|z^t, u^t) = \eta P(z_t|x_t \int P(x_t, m|u_t, x_{t-1}, m)P(x_{t-1}, m|z^{t-1}, u^{t-1})dx_{t-1}dm \qquad (2.4)$$

Este filtro propuesto y definido por la ecuación 2.4 no es sencillo de calcular por lo que es preciso realizar simplificaciones o suposiciones que faciliten su uso.

2.3.1.2. Filtro de Kalman (KF)

En 1960, Rudolf Emil Kálmán [10] publicó un artículo que describía una solución recursiva al problema de filtrado lineal de datos discretos, esta solución se la conoce como *filtro de Kalman*.

El Filtro de Kalman es un tipo de de Filtro de Bayes y es esencialmente un conjunto de ecuaciones matemáticas que implementan una estimación de tipo *predicción-corrección* que minimiza la covarianza del error estimado cuando se cumplen ciertas condiciones. Desde el momento en que se desarrolló, el Filtro de Kalman ha sido sujeto a una extensa investigación y aplicación, particularmente en el área de navegación autónoma o asistida [18]. Los avances en computación digital son los que han hecho práctico el uso de este filtro, aunque su éxito también se debe a la simplicidad y su naturaleza robusta. Es raro que se den todas las condiciones necesarias para que el filtro sea realmente óptimo, pero aún así el filtro consigue funcionar adecuadamente.

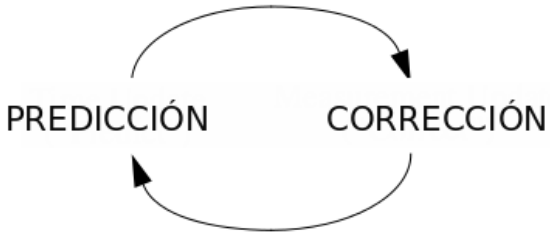


Figura 2.21: Ciclo continuo del Filtro de Kalman.

Para la comprensión y explicación del Filtro de Kalman nos hemos apoyado en el artículo escrito por Greg Welch y Gary Bishop [10], en el que proporcionan una introducción práctica de dicho filtro, una descripción y alguna discusión sobre el filtro de Kalman Extendido, acompañado de un ejemplo sencillo con números reales y resultados.

El Filtro de Kalman representa distribuciones a través de representaciones de momentos en el tiempo t: la distribución es representada por la media μ_t y la covarianza Σ_t . Los posteriores son Gaussianos para cualquier t, si se cumplen las siguientes tres propiedades, además de las hipótesis del filtro de Bayes 2.3.1.1:

- La probabilidad de la siguiente ecuación $p(x_t|u_t, x_{t-1})$ debe ser una función lineal con ruido aditivo Gaussiano. Esto viene dado por la siguiente ecuación:

$$x_t = Ax_{t-1} + Bu_{t-1} + \omega_{t-1} \tag{2.5}$$

  donde x_t es el vector de estados, u_t el vector de control, ω_t representan al ruido Gaussiano de media cero y covarianza Q del vector de estado. A y B son matrices que representan el modelo lineal que cumple el sistema en cuestión.

- La probabilidad de medida $p(z_t|x_t)$ debe ser también lineal y con ruido aditivo Gaussiano:

$$z_t = Hx_t + \delta_t \tag{2.6}$$

  donde H es una matriz que representa el modelo de observación del sensor. El vector δ_t representa al ruido Gaussiano de medición con media cero y covarianza R del vector z_t.

- La distribución inicial debe ser una distribución normal.

Partiendo de un sistema modelado con las ecuaciones 2.5 y 2.6, el filtro de Kalman consiste en un proceso continuo de *predicción-corrección*. Así pues, en la fase de predicción (actualización temporal), se obtiene una estimación a priori del siguiente instante de tiempo. La fase de corrección (actualización de medidas), se realiza cada vez que llega una medida, incorpora la información procedente de la medida a la estimación a priori para obtener una estimación mejorada. En la figura 2.22 se puede ver el diagrama del filtro de Kalman.

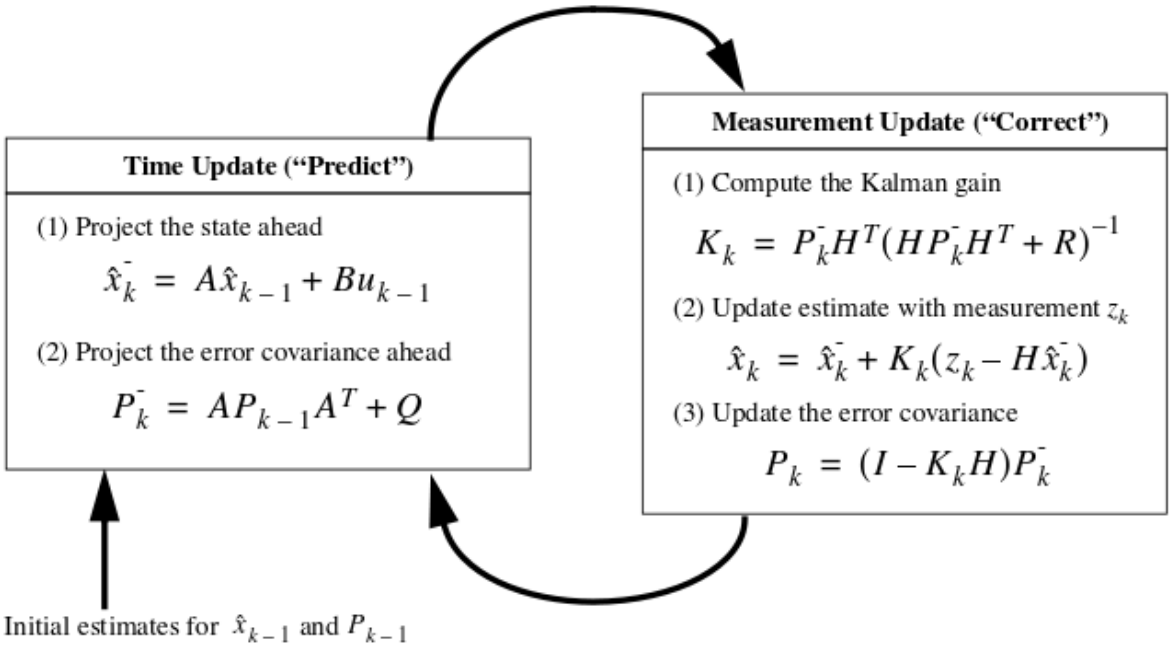


Figura 2.22: Diagrama de funcionamiento del Filtro de Kalman.

2.3.1.3. Filtro de Kalman Extendido (EKF)

Derivados del desarrollo previo del Filtro de Kalman, el Filtro de Kalman Extendido resuelve el problema de la no linealidad en el modelo de la *pose* del robot. Por ejemplo, un

robot que se mueve con una velocidad lineal y de rotación constantes describirá un círculo que no podrá ser descrito mediante transiciones de estado lineales. Esto hace al Filtro de Kalman inaplicable para la mayoría de problemas triviales en robótica

Este filtro para superar la hipótesis de linealidad, supone que las probabilidades del estado siguiente y de la medida están gobernadas por funciones no lineales f y h, respectivamente:

$$x_t = f(u_t, x_{t-1}) + \omega_t \tag{2.7}$$

$$z_t = h(u_t, x_{t-1}) + \delta_t \tag{2.8}$$

Este modelo generaliza el modelo lineal Gaussiano de los filtros de Kalman, postulado en las ecuaciones 2.5 y 2.6. La función f reemplaza a las matrices A y B en 2.5, y h reemplaza a la matriz H en 2.6. Desafortunadamente, con cualquier función lineal f y h, la estimación no será Gaussiana, de hecho es imposible para funciones no lineales conseguir una estimación exacta.

El Filtro de Kalman Extendido calcula una aproximación de la verdadera distribución de estimación. Representa dicha aproximación con una Gaussiana, representada por su media y su varianza. De esta manera este filtro puede funcionar igual de bien que el Filtro de Kalman, con la diferencia de que la distribución de estimación no es exacta, sino aproximada.

La idea clave bajo el Filtro de Kalman Extendido es la linealización. Dadas unas funciones no lineales f y h, se obtendrán las linealizaciones de las mismas por expansión de Taylor. El resultado es una matriz conocida como Jacobiano que viene dado por la siguiente ecuación, que se obtiene de calcular los gradientes de f y h, en un punto determinado.

$$A = \frac{\partial f}{\partial x}, H = \frac{\partial h}{\partial x} \tag{2.9}$$

Además de estos dos jacobianos, se suelen añadir dos más que dependen del ruido, por si éste tampoco es lineal:

$$W = \frac{\partial f}{\partial \omega}, V = \frac{\partial f}{\partial \delta} \tag{2.10}$$

A continuación, se puede ver el funcionamiento del algoritmo del filtro de Kalman extendido y las ecuaciones necesarias para llevarlo a cabo. Obsérvese como se usan por un lado las funciones no lineales y por otro los jacobianos de las mismas.

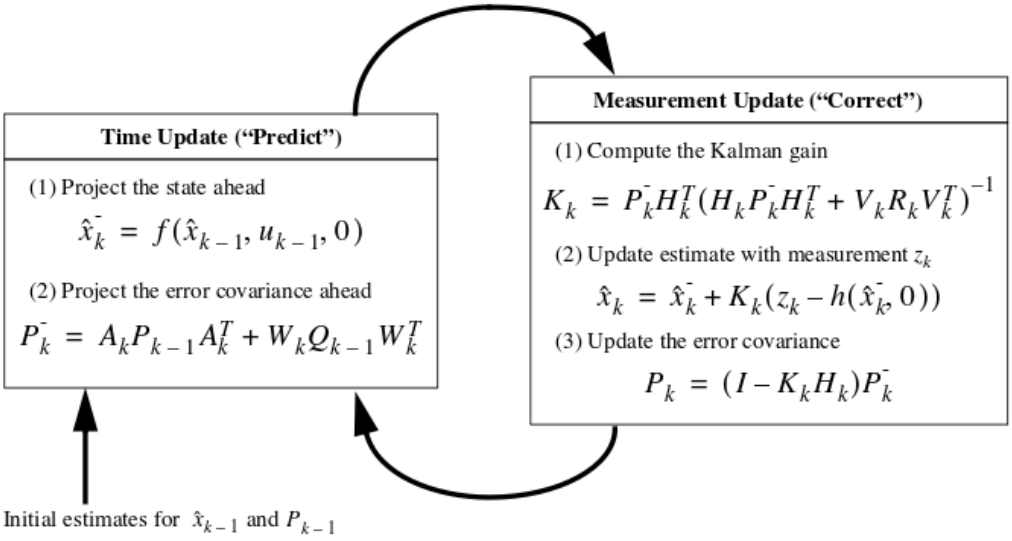


Figura 2.23: Diagrama de funcionamiento del Filtro de Kalman Extendido.

A pesar de la ventaja mencionada anteriormente, esta solución sufre tres importantes limitaciones:

- La complejidad para cada actualización, el número de características que describe el mapa, a priori es desconocido, por lo que si el tamaño del entorno a modelar es grande, a medida que la ruta del robot es mayor, la actualización se va complicando cada vez más.
- No pueden incorporar información negativa, es decir, no pueden usar el hecho de que un robot pueda no ver una característica aunque ésta sea esperada.
- No proveen soluciones al problema de asociación de datos. La falsa asociación de datos conduce frecuentemente a errores catastróficos en el mapeado.

Un conjunto de pruebas realizadas basadas en este filtro [19], en relación al problema SLAM 2D, dice que cuando los modelos estadísticos son conocidos, el algoritmo del Filtro de Kalman Extendido es una solución sólida. Sin embargo, esta situación no ocurre en la mayoría de las situaciones.

2.3.1.4. Filtro de Partículas (PF)

El filtro de partículas [20] es una implementación alternativa no paramétrica del filtro de Bayes. La idea clave del PF es representar la distribución de predicción con un conjunto de muestras de estado aleatorias de dicha distribución en lugar de con parámetros, como la media y la covarianza. Esta representación es aproximada, pero al ser no paramétrica es capaz de representar muchísimas más distribuciones que la Gaussiana.

En un filtro de partículas, las muestras de una distribución de predicción se llaman partículas y se denotan como:

$$\chi_t = x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]} \tag{2.11}$$

Cada partícula $x_t^{[m]}$ (con $1 \leq m \leq M$) es una muestra concreta del espacio en el instante t, esto es, una hipótesis de cómo debería ser el mundo real en el instante t. Aquí M

denota el número de partículas que forman el conjunto χ_t . En la práctica, el número de partículas M suele ser un número grande (por ejemplo, $M = 1000$), aunque en algunas de las implementaciones M es funciún de t o de algún otro parámetro.

La idea del PF es aproximar la distribución de estimación de x_t con el conjunto de partículas χ_t . De esta manera, cuantas más partículas haya más probabilidad habrá de que el estado verdadero está en la zona cubierta de partículas, es decir, más se parecerá la distribución estimada por las partículas a la real.

Como otros filtros de Bayes, este algoritmo también hace una estimación recursiva desde la estimación en el instante anterior. Para el caso de las partículas, este filtro construye un conjunto de partículas χ_t recursivamente en base al conjunto obtenido en el instante anterior x_{t-1}. Las entradas del algoritmo son el conjunto de partículas x_{t-1} además del orden de control u_t y la medida z_t, más recientes. El filtro construye un conjunto temporal de partículas $\bar{\chi}$, y lo hace $[m]$ procesando sistemáticamente cada partícula $x_{t-1}^{[m]}$ del conjunto χ_{t-1} como sigue:

- Se genera un estado hipotético $x_t^{[m]}$ para el instante t en base a la partículas $x_{t-1}^{[m]}$ y el control u_t. Este paso contiene el muestreo de la distribución del estado siguiente, esto es como una predicción de donde estará cada partícula en el siguiente instante de tiempo basándose en el estado anterior y en el control.
- Después se realiza el muestreo (importance sampling). Aquí se actualiza el peso $w_t[m]$ de cada partícula (importance factor). Este factor de peso se usa para incorporar las medidas z_t en el conjunto de particulas. El peso es la probabilidad de que la medida z_t se corresponda con el estado de la partícula $x_t[m]$, esto es: $w_t[m] = P(z_t|x_t[m])$. El conjunto de partículas con sus correspondientes pesos representará la estimación de predicción del estado.
- El remuestreo es la parte más importante del algoritmo, reemplaza M partículas del conjunto temporal $\bar{\chi}$. La probabilidad de que una partícula aparezca en el nuevo conjunto temporal viene dada por el peso de la misma. El nuevo conjunto tendrá el mismo tamaño, por lo que habrá partículas que no aparezcan en este nuevo conjunto (las que tenían un peso menor, y por tanto, menos probabilidad de entrar) de la misma manera que habrá partículas duplicadas (las de mayor peso, las más probables).

El remuestreo tiene la importante función de forzar a las partículas a ir hacia la estimación de predicción, es decir, hacia el estado real, abandonando las hipótesis menos probables. Sin esta fase, el proceso de convergencia a una solución sería mucho más lento, y en cualquier caso, menos preciso por malgastar partículas en zonas prácticamente improbables.

Como se ha comprobado, existen diferentes tipos de filtros. Para este proyecto es necesario el uso de un algoritmo que contenga una baja carga computacional y tenga un filtro de estimación posición robusto. Por ello, el uso del algoritmo Hector-Mapping está justificado ya que proporciona un EKF que predice la posición 3D del robot a partir de la lectura de datos de diferentes sensores y es un algoritmo que tiene una carga computacional baja adaptándose perfectamente a las necesidades de este proyecto. Por último, se han tenido en especial consideración los siguientes tres puntos que caracterizan a este algoritmo:

- Está implementado en ROS.
- No requiere odometría en el modelo de predicción, por lo que es apropiado para MAVs.
- Es ampliamente utilizado en la actualidad por múltiples grupos de investigación en robótica.

Capítulo 3

Algoritmo Hector-SLAM

Hector[1] es un equipo de investigación de la *Universidad Técnica de Darmstadt* establecido a finales del 2008. El algoritmo que en este capítulo se describirá, ha sido creado por Stefan Kohlbrecher para resolver el problema SLAM utilizando un sistema obtención online de mapas basados en rejillas con una carga computacional baja, combinando un **LIDAR**[2] y un sistema de estimación 3D basado en la información de diferentes sensores.

La habilidad de un modelo robótico de aprender el entorno en el que se encuentra y localizarse a sí mismo en dicho entorno es una de las habilidades más importantes que los robots autónomos son capaces de operar en el mundo real [11]. Teniendo en cuenta esta habilidad, el algoritmo Hector-Mapping está desarrollado para poder obtener un mapa 2D y representar una localización del robot, así resolviendo el problema del SLAM, basándose en la información de diferentes sensores y sin necesidad de utilizar odometría. Este algoritmo se ha implementado sobre todo en UGV's (Vehículo Terrestres no Tripulado), pero también es apropiado su uso en MAV's.

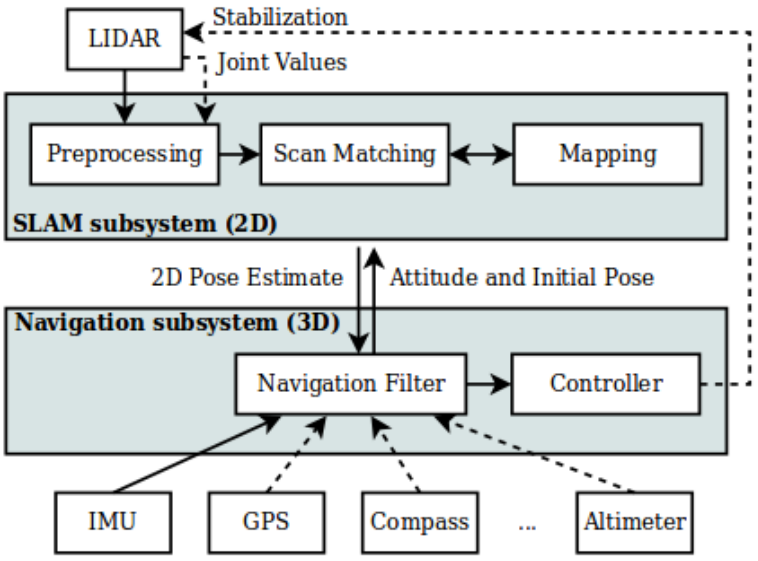


Figura 3.1: Sistema de mapeao y navegación en el que se basa el algoritmo Hector-Mapping.

[1] **He**terogeneous **C**ooperating **T**eam **o**f **R**obots

[2] Sensor láser. En este proyecto se ha utilizado un sensor Hokuyo URG-04LX-UG01

La mayoría de los enfoques de 2D SLAM requieren información de odometría, esto hace que no se aprovechen la alta velocidad establecida por los sensores LIDAR. Este algoritmo se centra en la estimación del movimiento del robot en tiempo real a través de **scan-matching** 3.1, haciendo uso de la alta frecuencia de actualización y el bajo ruido de las mediciones obtenida por los LIDAR. Este enfoque no proporciona la capacidad de cierre de lazo. Por ello, este algoritmo está pensando para su utilización en entornos pequeños donde no es necesario la utilización de cierre de lazos y entornos donde los sensores LIDAR son beneficiosos.

El algoritmo Hector-Mapping estima un estado 6DOF es decir, movimiento en los 3 ejes combinando la traslación y la rotación. Con este algoritmo se obtiene un mapa 2D en el que se actualizan los ángulos (x, y, Ψ) y el estimador de posición se actualizan los ángulos $(x, y, z, \Theta, \Phi, \Psi)$, obteniendo así una estimación de posición 6DOF. Para no entrar en confusión, en este proyecto se hablará de estimación 6DOF como estimación 3D.

Por lo tanto, este algoritmo realiza una estimación 3D basándose en las rotaciones y traslaciones de la plataforma robótica. Para lograr esto, un filtro de navegación fusiona la información obtenida de la IMU y de los otros sensores (ultrasonidos, LIDAR, etc) para así obtener una estimación sólida en 3D, mientras que el sistema 2D SLAM es usado para proporcionar una estimación y una información de la trayectoria del MAV a nivel del suelo. Ambas estimaciones se actualizan individualmente y tan sólo están unidas para permanecer sincronizadas con el tiempo.

3.1. 2D SLAM

La representación de un entorno mediante el uso de este algoritmo se basa en la utilización de los puntos finales de la nube de puntos obtenida a partir del LIDAR. Estos puntos son representados, con unas coordenadas cartesianas X e Y, en el plano del láser. Mediante el uso de estos puntos, el algoritmo los procesa obteniendo así un mapa de ocupación de rejillas, donde los colores grises representan espacios vacíos y el color negro los muros u objetos fijos dentro del entorno.

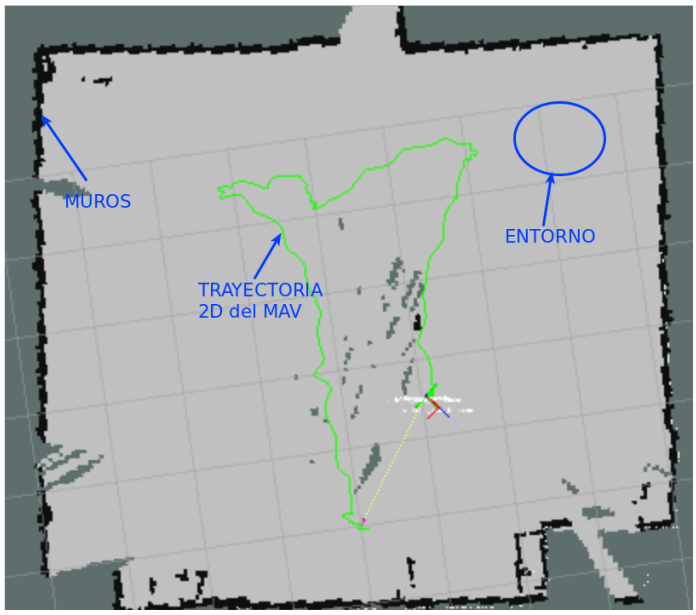


Figura 3.2: Ejemplo mapa de rejilla mediante el algoritmo Hector-Mapping

El *scan-matching* es el proceso de alineación del láser entre sí o con un mapa existente. Este enfoque de "alineación" está basado en el trabajo de visión computacional [12]. Mediante este enfoque, no es necesario una búsqueda de asociación de datos entre los puntos finales del haz del LIDAR o una búsqueda exhaustiva del "pose". Como el "scan" consigue alinearse con el mapa existente en el tiempo, el "matching" se lleva a cabo de manera implícita con todos los "scans" anteriores. El algoritmo utiliza un enfoque de Gauss-Newton para la optimización y formula la transformación rígida mediante la siguiente ecuación:

$$\xi^* = argmin_\xi \sum_{i=1}^{n} [1 - M(S_i(\xi))]^2 \tag{3.1}$$

La ecuación 3.1 permite encontrar el mejor alineamiento del haz del láser (nube de puntos) con el mapa, es decir, la posición de cada punto en el mapa de rejillas creado. Donde $S_i(\xi)$ son las coordenadas en el "mundo" de la nube de puntos o puntos finales $s_i{=}(s_{i,x},\ s_{i,y})^T$. Estas coordenadas son función de ξ, donde ξ es la transformación rígida de la nube de puntos proyectada en el mapa obtenido $\xi = (p_x, p_y, \Psi)^T$. Las coordenadas del robot en el mapa (pose del robot) se obtiene mediante la ecuación:

$$S_i(\xi) = \begin{pmatrix} \cos(\Psi) & -\sin(\Psi) \\ \sin(\Psi) & \cos(\Psi) \end{pmatrix} \begin{pmatrix} s_{i,x} \\ s_{i_y} \end{pmatrix} + \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{3.2}$$

Mediante la ecuación 3.2 se obtienen las coordenadas de los puntos en el mapa y la función $M(S_i(\xi))$ devuelve el valor del mapa a partir de las coordenadas obtenidas por $S_i(\xi)$. Dada una estimación inicial de ξ, es necesario optimizar el error de medida $\Delta\xi$ mediante:

$$\Delta\xi = H^{-1} \sum_{i=1}^{n} [\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}]^T [1 - M(S_I(\xi))] \tag{3.3}$$

Con la matriz *Hessiana*[3] :

$$H = [\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}]^T [\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}] \tag{3.4}$$

La ecuación 3.3 encuentra la transformación que mejor se adapta al haz del láser para así reducir el error que se genera al estimar una posición de los puntos del láser al proyectarlos en el mapa dada por la ecuación 3.1. Donde H es una aproximación para el gradiente del mapa $\nabla M(S_i(\xi))$.

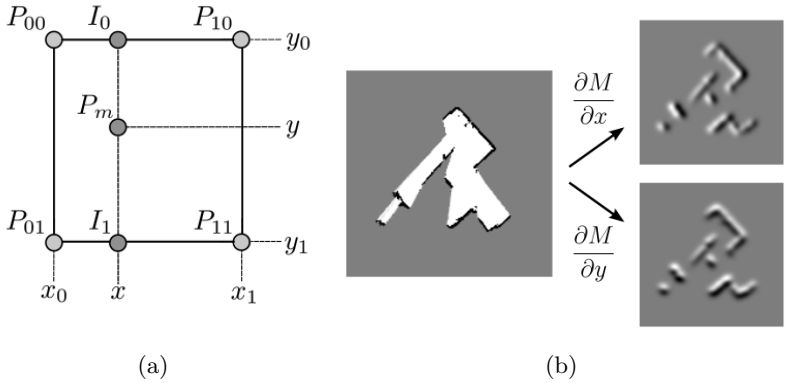
Las ecuaciones 3.3 y 3.4 se basan en el gradiente del mapa $\nabla M(P_m)$. El gradiente se puede aproximar por medio de filtrado bilineal de los cuatro enteros más cercanos de las coordenadas en el mapa P_{00}, P_{01}, P_{10} y P_{11}, que representan los puntos medios de las celdas del mapa de P_m, ver figura 3.3.

$$\frac{\partial M}{\partial x}(P_m) \approx \frac{y - y_0}{y_1 - y_0}(M(P_{11}) - M(P_{01})) + \frac{y_1 - y}{y_1 - y_0}(M(P_{10}) - M(P_{00})) \tag{3.5}$$

$$\frac{\partial M}{\partial y}(P_m) \approx \frac{x - x_0}{x_1 - x_0}(M(P_{11}) - M(P_{10})) + \frac{x_1 - x}{x_1 - x_0}(M(P_{01}) - M(P_{00})) \tag{3.6}$$

[3]En Matemática, la matriz hessiana o hessiano de una función f de n variables, es la matriz cuadrada de nxn, de las segundas derivadas parciales.

Donde x, y, x_0, x_1, y_0 e y_1 son las coordenadas del mapa de los valores enteros P, como se muestra en la figura 3.3.



(a) (b)

Figura 3.3: La figura (a) representa el filtro bilineal del mapa de ocupación por regillas. El punto P_m, es el punto cuyo valor es interpolado. La figura (b) es un ejemplo de representación del mapa y sus derivadas parciales.

Para muchas aplicaciones es deseable una aproximación Gaussiana de la incertidumbre al proceso del "match". Ejemplos de ello son las actualizaciones de filtros paramétricos y el uso de la matriz de Hesse (matriz H, Hessiana) aproximada para llegar a una estimación de covarianza, donde la matriz de covarianza se aproxima por $R = \sigma^2 H^{-1}$ y σ es un factor escalable que depende de las propiedades del sensor láser [13].

Otra optimización del proceso scan-matching se logra a través de otro enfoque inspirado en visión artificial y procesamiento de imágenes. Dado que el procedimiento se basa en el descenso del gradiente, quedarse atrapado en mínimos locales es una fuente de error. Hector-SLAM busca minimizar estos efectos utilizando una estructura jerárquica de imagenes, donde el mapa se almacena en diferentes resoluciones, los cuales se mantienen en concordancia. Un efecto secundario es la disponibilidad inmediata de los mapas, que pueden ser usados, por ejemplo, para la planificación de trayectorias.

(a) (b) (c)

Figura 3.4: Representación de mapa con diferentes resoluciones: (a) Resolución con longitud de regilla 20cm, (b) Resolución con longitud de regilla 10cm y (c) Resolución con longitud de regilla 5cm.

3.2. 3D State Estimation

Para la estimación de la posición 3D de la plataforma robótica, el algoritmo Hector-Mapping utiliza un filtro de Kalman (ver sección 2.3.1.2) mediante las ecuaciones del modelo robótico definido por:

$$\dot{\Omega} = E_\Omega \cdot \omega \tag{3.7}$$

$$\dot{p} = v \tag{3.8}$$

$$\dot{v} = R_\Omega \cdot a + g \tag{3.9}$$

Donde R_Ω es la matriz coseno director que asigna un vector en la referencia base del drone (*body frame*) para la referencia de navegación (*navigation frame*). E_Ω representa las velocidades angulares de los ángulos de Euler y g es la constante de gravedad. Las ecuaciones 3.7, 3.8 y 3.9 son ecuaciones diferenciales no lienales obtenidas a partir de:

$$\Omega = (\phi, \theta, \psi)^T \tag{3.10}$$

$$p = (p_x, p_y, p_z)^T \tag{3.11}$$

$$v = (v_x, v_y, v_z)^T \tag{3.12}$$

Donde ϕ, θ y ψ son los ángulos de Euler *roll*, *pitch* y *yaw* respectivamente, el vector p de posición y v el vector de velocidad.

Por otro lado, el vector de estado varía con la variación de los giróscopos y acelerómetros ya que varían en el tiempo e influyen en el resultado de manera significativa. El uso del filtro de Kalman Extendido está justificado ya que el sistema de ecuaciones (ecuaciones 3.7, 3.8 y 3.9) es un sistema no lineal debido al uso de los ángulos de Euler usados en las matrices E_Ω y R_Ω.

La posición y la velocidad se calcula mediante una integración de las aceleraciones y se realimenta para que el sistema no sea inestable cada vez que se va estimando la velocidad y la posición nueva.

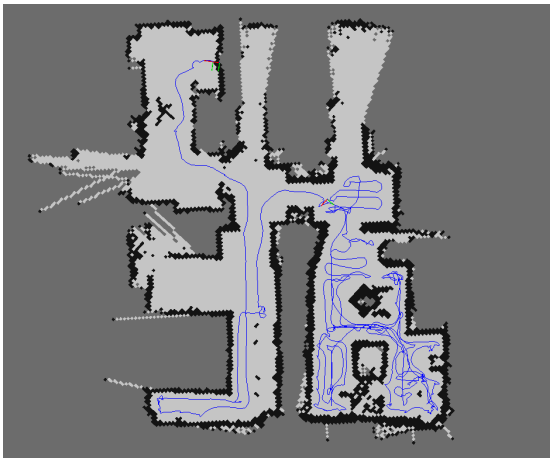


Figura 3.5: Mapa con trayectoria basado en el algoritmo Hector-Mapping

En definitiva, el algoritmo Hector-Mapping se puede usar en diversos escenarios como en simulaciones para la búsqueda y rescate humano usando plataformas robóticas, mapeo

litoral en un USV[4], etc. El sistema estima e incorpora correctamente un estimador 3D de posición mediante el uso de un LIDAR con una carga computacional baja. Aunque el algoritmo Hector-Mapping incorpora un estimador de posición 3D, a lo largo del proyecto se ha optado por crear un estimador de posición aparte ya que la lectura de los datos del MAV (AR Drone 2.0) no ha sido fácil integrarla en el algoritmo, obteniendo resultados incorrectos.

[4]Unmanned surface vehicles

Capítulo 4

Desarrollo

En este capítulo se describirán las herramientas principales utilizadas en este proyecto, tales como las plataformas robóticas, sensores y sistemas de comunicación. Se hará una descripción detallada de la arquitectura general del sistema en la cual se describirá la configuración del MAV utilizado y por último, se adjuntarán resultados obtenidos en simulación y se detallarán los pasos necesarios para la implementación sobre la plataforma real.

4.1. Plataforma robótica

En esta sección se describirá el hardware y el software utilizado, centrándose en especial en las especificaciones de diseño de la plataforma robótica que se ha utilizado. En la figura 4.1 se puede observar la plataforma real.

Figura 4.1: Estructura completa con los sensores montados en el robot real. En rojo la Raspberry Pi, azul los ultrasonidos y verde el sensor láser Hokuyo.

4.1.1. AR Drone 2.0 de Parrot

Como se ha comentado en la sección 2.1.4, la plataforma robótica elegida para la realización de este proyecto ha sido AR Drone 2.0 del fabricante Parrot.

A continuación se detallarán las características y especificaciones del AR Drone 2.0.

Estructura, dimensiones y motores

El AR Drone 2.0 tiene **4 motores** de rotor interno sin escobillas controlados con un microcontrolador AVR de 8 MIPS. Estos motores tienen una potencia 14,5W y una velocidad nominal de 2800 rpm.

Está construido con fibra de carbono, lo que hace que el AR Drone 2.0 pese un total de 380 gramos con el casco de protección para el exterior y 420 gramos con el casco de protección para el interior, incluyendo en estos pesos la batería recargable Li-Po de 3 elementos y 1.000 mA/h, la cual proporciona una autonomía de 12 minutos.

La siguiente tabla 4.1 hace una comparativa de las dimensiones del drone con y sin casco de protección.

	Dimensiones
Sin casco de protección	451x451 mm
Con casco de protección	517x517 mm

Tabla 4.1: Dimensiones del AR Drone 2.0 con y sin protección.

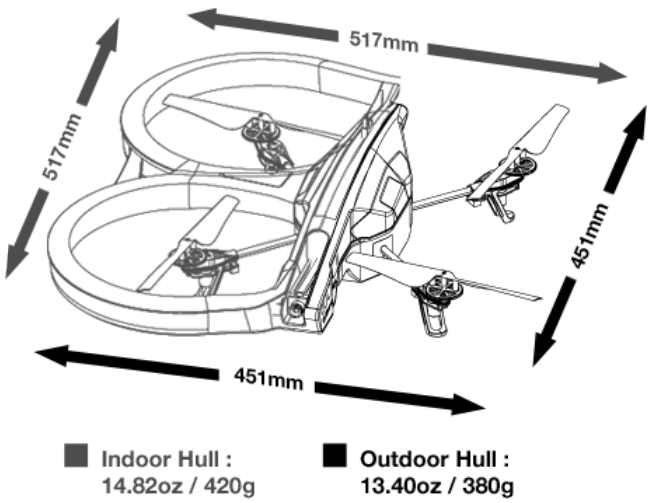


Figura 4.2: Dimensiones y peso del AR Drone 2.0.

Sensores

AR Drone 2.0 cuenta con diferentes sensores integrados que permiten obtener diferentes medidas necesarias en los MAVs. La medición de altura respecto al suelo se obtiende mediante el uso de un **ultrasonidos** que se encuentra en la base del drone. Cuando se supera la altura máxima a la que el ultrasonidos puede medir, utiliza un barómetro incorporado en él para medir la presión atmosférica y calcular la altura a partir de ella.

Incoropora un **acelerómetro de 3 ejes** con el cual se puede obtener la velocidad del drone respecto a los 3 ejes, "*X, Y y Z*". A parte del acelerómetro, utiliza una cámara vertical QVGA implementada en la base del drone para medir la velocidad del drone respecto al suelo.

La orientación del drone es calculada mediante el **giróscopo** y el **magnetómetro** que incorpora en la placa. Mediante estos sensores puede medir cuánto ha girado el drone respecto la posición inicial, cuánto necesita girar y si ha girado demasiado para la estabilización del mismo.

La tabla 4.2 muestra la información de los sensores que incorpora el AR Drone 2.0.

Sensores	Información
Cámaras	Cámara frontal HD a 720p. Cámara vertical QVGA. Es utilizada para el cálculo de la velocidad respecto al suelo y la estabilidad del drone.
Giróscopo de 3 ejes	Utilizado para estabilizar el drone. Precisión: $2000^o/s$
Magnetómetro de 3 ejes	Utilizado para medir la dirección del campo magnético de la tierra y junto al acelerómetro, calcula la orientación del drone. Precisión: 6^o
Acelerómetro de 3 ejes	Utilización de un acelerómetro de 3 ejes para el cálculo de velocidad en eje X Y Z. Precisión: $\pm 5mg$
Barómetro	Necesario para el cálculo de presión atmosférica. Precisión a 80 cm a nivel del mar: $\pm 10Pa$
Ultrasonidos	Utilizado para la medición de altura del drone respecto al suelo. Alcance máximo: 6 metros

Tabla 4.2: Características de los sensores del AR Drone 2.0.

Procesador

La tecnología integrada en el AR.Drone 2.0 permite un control de precisión extremo y características de estabilización automática mediante la CPU y el trabajo en conjunto de los diferentes sensores que incorpora.

Incorpora un mini ordenador que cuenta con un **CPU**[1] **ARM Cortex A8 de 32 bits** a 1 GHz y con DSP[2] TMS320DMC64x a 800 MHz con una memoria **RAM DDR2 de 1GB a 200 MHz** y sistema operativo **Linux 2.6.32**. Además, cuenta con **Wi-Fi b/g/n** y con un **USB 2.0** a alta velocidad que permite obtener los vídeos grabados con la cámara del drone y conectar extensiones como un GPS.

Mediante Linux, el "mini ordenador" permite controlar los motores del drone en función de la información de los sensores y además sostiene el router Wi-Fi para que los dispositivos móviles puedan conectarse como lo hacen con cualquier *Access Point* para trasladar las acciones que se le soliciten desde la aplicación *AR. FreeFlight* o desde el entorno robótico ROS.

En definitiva, como punto favorable tras analizar estas especificaciones de la plataforma robótica AR Drone 2.0 elegida, se ha concluido que AR Drone 2.0 cumple con las características necesarias para este proyecto. Por lo contrario, al tratarse de un dispositivo creado

[1]Siglas en inglés: Central Processing Unit
[2]Siglas en inglés: Digital Signal Processor

para nivel recreativo y no para investigación propiamente dicha, el drone no puede soportar pesos muy elevados y esto puede suponer un problema a la hora de implementar los diferentes sensores necesarios para realizar el SLAM (como el caso de Hokuyo) por lo que este punto hay que tenerlo muy presente para poder aligerar el drone o ver las diferentes opciones de sensores que se pueden incluir en su hardware.

4.1.2. Sensor de barrido láser Hokuyo URG-04LX-UG01

El sensor de barrido láser elegido ha sido el Hokuyo URG-04LX-UG01[3], el cual ha sido diseñado bajo estándares JISC8201-5-2 y IEC60947-5-2 para aplicaciones industriales (ver figura 4.3). Se trata de un sensor láser para escanear áreas donde la fuente de luz del sensor es un diodo semiconductor láser infrarrojo de 785nm de longitud de onda con seguridad de "clase 1", es decir, el sensor es seguro para cualquier condición de funcionamiento y no requiere medidas de seguridad. Hay que tener en cuenta que para un correcto funcionamiento del sensor ha de ser utilizado sólo en interiores.

Figura 4.3: Sensor láser Hokuyo URG-04LX-UG01.

Este sensor tiene un barrido semicircular de 240º con un radio máximo de 4 metros de alcance, lo que es más que suficiente para detectar objetos a una distancia desde la plataforma robótica en la que es instalado.

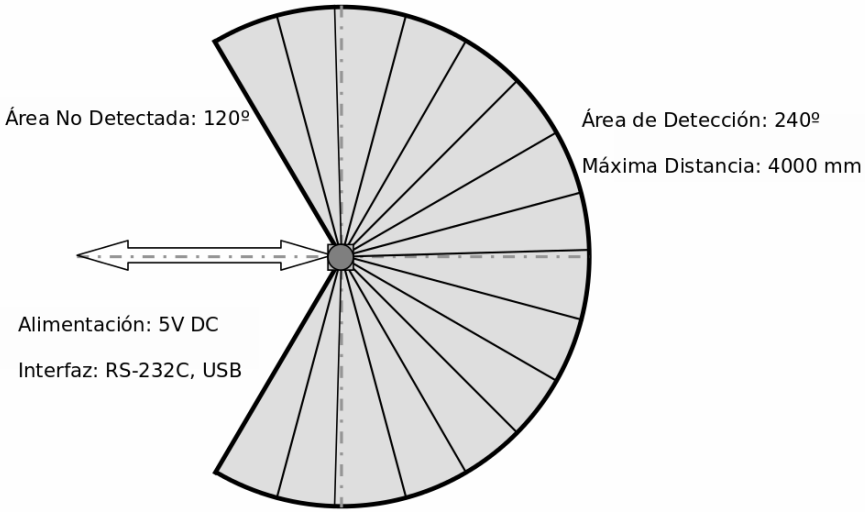


Figura 4.4: Rango del láser Hokuyo.

[3]http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx/

En la figura 4.4 muestra el área detectable sin ningún objeto detectado. Esta distancia puede variar según el objeto detectado.

Algunas características importantes que se han tenido en cuenta para este proyecto se declaran en la tabla 4.3:

Especificaciones	Datos numéricos
Alimentación	5 Voltios
Exactitud en la distancia	20mm - 1000mm $\pm 10mm$ 1m - 4 metros $\pm 1\,\%$
Resolución de distancia	1 mm
Ángulo de escaner	240^o
Resolución de ángulo de escaner	0.36^o
Peso	160 gramos
Dimensiones	50x50x70 mm
Material de diseño	Policarbonato
Velocidad angular	360 grados/s
Aceleración angular	$\Pi/2$ rad/s^2
Nivel de Ruido	25db o menos a 300 mm
Vida Útil	5 años. Dependiendo de las condiciones de uso

Tabla 4.3: Características del Hokuyo URG-04LX-UG01.

Debido a que Hokuyo cuenta con unas características considerablemente buenas, es de dimensiones pequeñas y que su implementación en ROS es directa, ya que cuenta con un paquete de drivers (ver sección 4.1.4) en este entorno de desarrollo el cual permite utilizar el sensor en cualquier aplicación robótica y gracias a él se accede a los datos del sensor facilmente, se ha decidido utilizar este dispositivo para obtener los datos del entorno y así poder crear un mapa en 2D a partir de la nube de puntos generada por él al detectar los objetos. Otra de las razones que se ha tenido en especial consideración a la hora de elegir este sensor, ha sido la posibilidad de utilizar este sensor con el algoritmo Hector Mapping, el cuál utilizada la nube de puntos generada por el sensor para crear un mapa 2D.

4.1.3. Elementos adicionales

En este apartado, se describirán brevemente los elementos adicionales que se han necesitado incorporar al hardware del drone para poder realizar la comunicación entre éste y el portátil y otros senores que son necesarios para otras aplicaciones. Hay que aclarar que los algoritmos utilizados en este apartado y la comunicación entre el drone y los sensores, es externa a este proyecto pero que es necesario comentar ya que estos elementos están implementados en el hardware del drone.

Para poder recibir los datos desde el Hokuyo incorporado en el drone, se necesita un "puente" que ayude en esta comunicación. Por ello, se necesita un ordenador que esté igualmente incorporado en el drone y procese los datos del sensor láser y los envíe al portátil donde tenemos el algoritmo SLAM.

En la parte de comunicación se ha optado por la utilización de un mini ordenador de-

nominado **Raspberry Pi 2 Model B**[4] (ver figura 4.5). Este mini ordenador utiliza un procesador ARMv7, y gracias a él, puede ejecutar el sistema operativo Linux que se utiliza en este proyecto. Al poder utilizar Linux, se ha instalado el entorno ROS en él junto con los paquetes necesarios, en este caso *hokuyo_node*. Así mismo se podrá crear algoritmos para la transmisión de los datos del láser Hokuyo al portátil, entre otros.

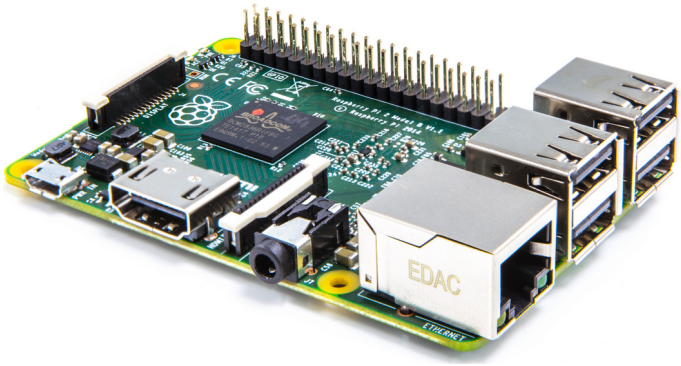


Figura 4.5: Raspberry Pi 2 Model B.

Por otro lado, se necesita que el drone evite el techo en interiores y para ello se utilizará, además de los sensores que previamente se han descrito, unos sensores de ultrasonidos que indiquen la distancia al techo y el suelo a la que se encuentra el drone para, mediante un algoritmo que se ha creado en ROS.

Figura 4.6: Ultrasonidos.

4.1.4. Paquetes de ROS

Para la realización de este proyecto se han utilizado diversos paquetes de ROS, desde paquetes de drivers que controlan los dispositivos hasta paquetes de algoritmos que son necesarios para el SLAM. Los siguientes apartados explican los diferentes paquetes que se han utilizado en este proyecto.

4.1.4.1. Paquetes de simulación

Para simular el sistema se ha utilizado el programa de simulación 3D "*Gazebo*" (ver sección 2.2.1.4) para simular un entorno de trabajo real y mediante el visualizador "*Rviz*" (ver sección 2.2.1.5) se podrá obtener una visualización en 2D del mapa del entorno a partir de la lectura del sensor láser implementado en el robot.

Los paquetes de los programas anteriormente descritos, se pueden encontrar en ROS como:

[4]https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

- **Gazebo:** El paquete en ROS se denomina **"gazebo_ros_pkgs"**[5] y se puede instalar de dos maneras, en un espacio de trabajo creado o se puede instalar a la vez que se instala el entorno de desarrollo ROS. En ocasiones es preferible instalarlo en un espacio de trabajo ya que de esta forma se puede acceder a los archivos que contiene el paquete y de la otra manera se instalará en el sistema y no se puede acceder a estos archivos, aunque seguirá funcionando de la misma manera.
- **Rviz:** El visualizador 3D Rviz se puede encontrar en el paquete **"rviz"**[6] y al igual que Gazebo, se puede instalar de las dos maneras anteriormente descritas.

Estos paquetes contienen los diferentes programas de simulación utilizados en este proyecto.

4.1.4.2. Paquetes de drivers

Algunos dispositivos necesitan de drivers específicos para la comunicación entre el dispositivo y el ordenador. Por ello, la comunicación entre el AR Drone 2.0 y el Hokuyo con el ordenador se basan en la utilización de unos paquetes disponibles en ROS.

- **AR Drone 2.0:** Para poder controlar y obtener las diferentes mediciones de los sensores del drone, se utiliza un paquete en ROS denominado **"ardrone_autonomy"**[7]. Este paquete contiene los drivers para los cuadricópteros AR Drone 1.0 y AR Drone 2.0 basándose en la SDK del AR Drone versión 2.0.1.

  Este paquete se encarga de la comunicación con el AR Drone y a su vez publica una serie de topics y servicios para su control y lectura de los sensores del mismo. Algunos de estos topics y servicios utilizados en este proyecto han sido:

  1. **ardrone/navdata:** Mediante este topic se pueden obtener distintas mediciones de los sensores del drone como el porcentaje de batería, velocidades, altura, rotaciones, entre otros. El paquete ardrone_autonomy permite la publicación de este topic y diferentes topics que permiten la lectura de los sensores en topics por separado. Para poder activar la publicación de estos topics por separado hay que habilitar su visualización, por ejemplo para activar la publicación del topic que permite la lectura del altímetro hay que habilitar el parámetro *enable_navdata_altitude* en un launch que previamente lance el paquete ardrone_autonomy con la configuración deseada.
  2. **ardrone/takeoff** y **ardrone/land:** Estos topics permiten el despegue y el aterrizaje del drone respectivamente. Para poder realizar el despegue o el aterrizaje es necesario publicar en dicho topic un mensaje vacío del tipo *std_msgs*.

     ```
     $ rostopic pub /ardrone/takeoff std_msgs/Empty " "
     ```

     ```
     $ rostopic pub /ardrone/land std_msgs/Empty " "
     ```

  3. **ardrone/flattrim:** Este servicio permite recalibrar la rotación del drone asumiendo que el AR Drone está en una superficie plana. Nunca se debe llamar a este servicio mientras el AR Drone está volando. El servicio se llama de tal manera:

     ```
     $ rosservice call /ardrone/flattrim " "
     ```

  Para obtener más información sobre este paquete se puede acceder a su páguina web[8].

[5] http://wiki.ros.org/gazebo_ros_pkgs
[6] http://http://wiki.ros.org/rviz
[7] http://wiki.ros.org/ardrone_autonomy
[8] http://ardrone-autonomy.readthedocs.org/en/latest/index.html

- **Hokuyo:** Hokuyo utiliza el paquete **"hokuyo_node"**[9] el cual permite la comunicación entre el ordenador y éste mediante la publicación del topic **scan** en el cual se publica la nube de puntos que se obtiene del láser Hokuyo. Este paquete ha de ser lanzado para la utilización del algoritmo *Hector Mapping* el cual utiliza la nube de puntos proporcionada por el Hokuyo para la representación de un entorno 2D. Para ello es necesario abrir el puerto mediante el siguiente comando:

```
$   sudo chmod a+rw /dev/ttyACM0
```

Una vez esté asegurado la apertura del puerto del láser, se debe lanzar el ejecutable del paquete:

```
$   rosrun hokuyo_node hokuyo_node
```

4.1.4.3. Paquetes de software

- **Transformaciones:** Para realizar transformaciones de coordenadas entre las distintas referencias (frames) de un robot, ROS contiene un paquete denominado **tf**[10]. Mediante este paquete el usuario puede hacer un seguimiento de múltiples frames para coordinarlos en el tiempo, es decir, obtener distintas transformaciones respecto a los diferentes frames que forman un modelo robótico (por ejemplo sus articulaciones como los brazos robots).tf mantiene la relación entre las coordenadas de los frames en una estructura de árbol controlado en el tiempo. Además, contiene una librería específica la cual permite al usuario transformar puntos, vectores, etc. entre coordenadas en el tiempo mediante el uso de ejecutables $C++$ o desde un terminal mediante el uso de "rostopic".

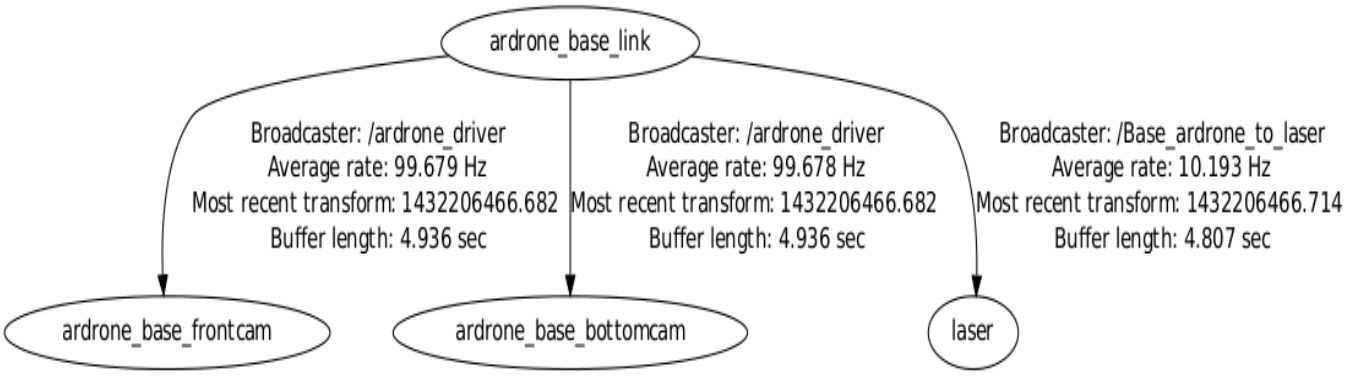


Figura 4.7: Ejemplo de árbol del paquete tf.

- **Estimador de posición 3D:** Uno de los paquetes probados para obtener una estimación 3D ha sido **robot_pos_ekf**[11]. Estima la posición 3D a partir de medidas leidas de diferentes fuentes de sensores. Este paquete usa un *filtro extendido de Kalman* (sección 2.3.1.2) para obtener un modelo de estimación 6DOF (3D de posición y 3D de orientación) a partir de la odometría del robot. Sin embargo, este paquete

[9] http://wiki.ros.org/hokuyo_node
[10] http://wiki.ros.org/tf
[11] http://wiki.ros.org/robot_pose_ekf

no se ha podido implementar en el proyecto debido a que se han obtenido resultados erróneos.

4.1.4.4. Paquetes de HECTOR:

- **Hector Quadrotor:** Se trata de una *pila* de paquetes que contiene paquetes de modelado, control y simulación de cuadricópteros. Mediante estos paquetes se puede simular entornos 3D, en Gazebo y Rviz, en los que mediante un modelo de un cuadricóptero se pueden sobrevolar dichos entornos y obtener una representación 2D del entorno mediante un algoritmo SLAM. Este algoritmo está presente en otra pila de paquetes denominada **Hector_slam**. Algunos de los paquetes más importantes usados en este proyecto han sido:
- **Hector_slam:** Esta pila de paquetes contiene todos los paquetes necesarios para utilizar los algoritmos SLAM (ver capítulo 3) para la obtención de mapas en 2D.
  1. **hector_mapping:** Este paquete contiene un algoritmo SLAM que puede ser usado sin odometría, muy útil para drones ya que carecen de ella. Utiliza información de sensores láseres acoplados a los robots para obtener mapas en 2D.
  2. **hector_geotiff:** Permite guardar los mapas 2D obtenidos en un formato especial "*GeoTiff*".

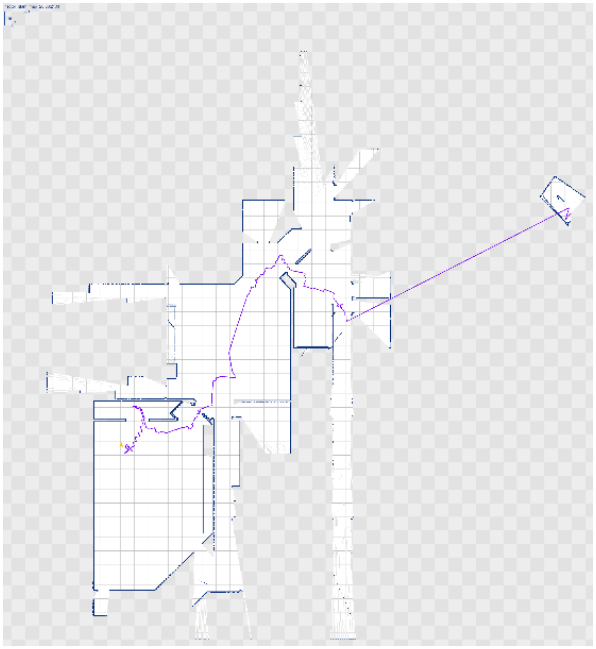


Figura 4.8: Mapa en formato geotiff.

  3. **hector_imu_attitude_to_tf**: Se trata de un nodo que puede ser utilizado para publicar los ángulos de posición "roll/pitch" reportados a través de un mensaje de IMU al tf.
  4. **hector_localization:** Esta pila de paquetes contienen todos los paquetes necesarios que proporcionan una estimación 3D de la *pose* de la plataforma robótica, en concreto se habla del paquete **hector_pose_estimation**. Utiliza para ello un filtro de Kalman Extendido a partir de sensores acoplados al robot. Este paquete no se ha podido implementar en este proyecto ya que los resultados obtenidos con estos paquetes han sido erróneos.

5. **message_to_tf:** Las transformaciones de coordenadas de los *frames* en ROS es importante. Por ello, Hector cuenta con este paquete que mediante la subscripción al topic *pose* puede obtener las transformaciones necesarias para crear un árbol de relaciones de transformaciones (ver figura 4.7) que son necesarias para la utilización de los paquetes de estimación de posición de Hector. Este paquete da como resultado una transformación en tres frames con frames intermedios *base_footprint* y *base_stabilized* (sin roll ni pitch) esto se puede ver en la imagen 4.21 y se explicará más adelante.

4.2. Simulación del sistema de localización y mapeado

La parte de simulación en este tipo de proyectos es muy importante ya que permite recrear un entorno real y obtener resultados para compararlos con la realidad. Así se puede obtener un sistema más preciso según las necesidades del proyecto y variar parámetros en el mismo según estos resultados obtenidos, antes de su implementación en el sistema real.

En esta sección se describirán los pasos necesarios para obtener un sistema de simulación del proyecto con un modelo en 3D del AR Drone 2.0 y un entorno de trabajo en Gazebo (sección 2.2.1.4) y Rviz (sección 2.2.1.5).

4.2.1. Modelado del AR Drone y los sensores

Los siguientes apartados describen los pasos y programas necesarios para implementar un modelo 3D en ROS del sistema a simular.

4.2.1.1. Modelado del AR Drone 2.0

Para la representación en 3D del robot se ha utilizado el programa de diseño CAD[12] *SolidWorks* versión 2013, desarrollado en la actualidad por SolidWorks Corp, el cual permite un modelado mecánico en 3D.

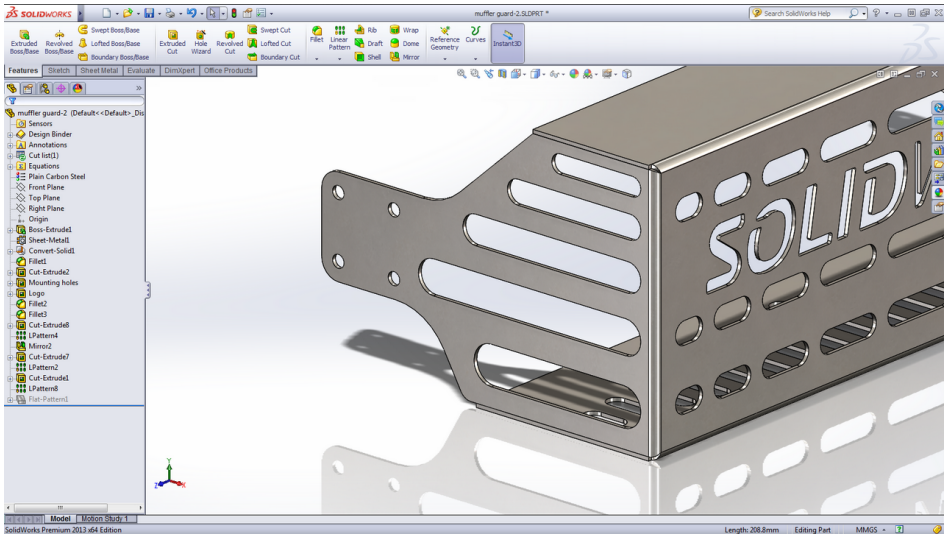
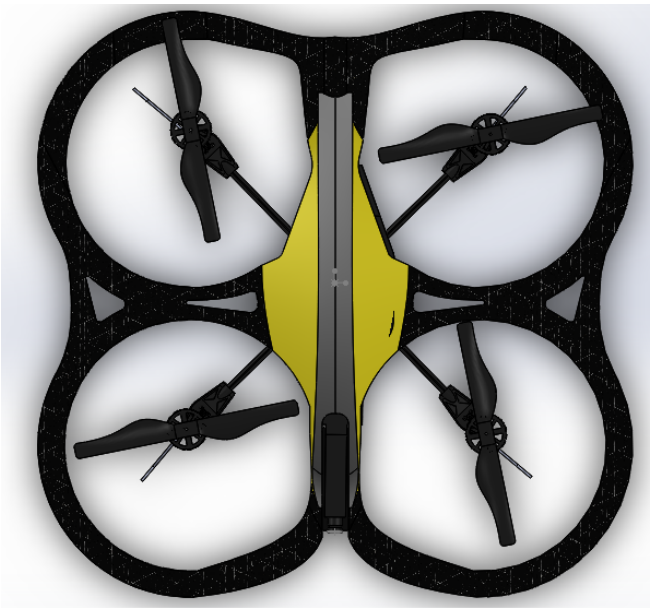


Figura 4.9: Captura de interfaz del programa CAD SolidWorks.

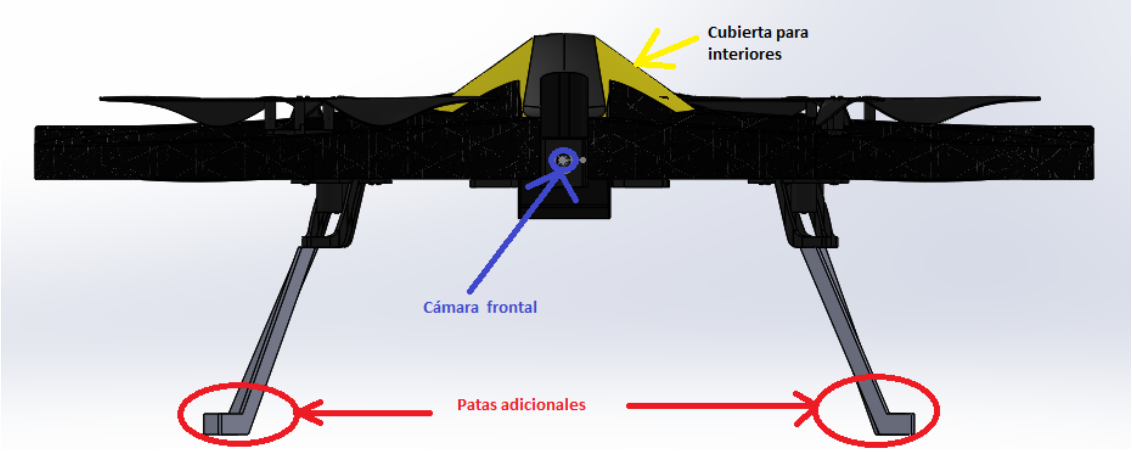
[12]Computer-Aided Design en inglés o Diseño Asistido por Computadora en español.

Hoy en día muchos modelos en 3D o planos en 2D se pueden encontrar en internet en diversos formatos CAD. Esto reduce mucho el tiempo de trabajo ya que algunos modelos en ocasiones son de elevada complejidad.

En la figura 4.10 se pueden observar las diferentes vistas del modelo en 3D del AR Drone 2.0. En ellas se pueden observar las diferentes partes del AR Drone y se ha decidido realizar el modelo 3D con la protección para interiores ya que este proyecto tiene como objetivo final una exploración en entornos de interior.



(a) Vista AR Drone Planta.



(b) Vista AR Drone Frontal.

Figura 4.10: Vistas AR Drone 2.0.

Por otro lado, en la figura 4.11 se observa la plataforma que se optará por usar en la simulación sin el modelo del sensor láser Hokuyo.

Figura 4.11: Modelo en 3D del AR Drone 2.0 usando SolidWorks.

Los controladores de vuelo del modelo creado se deben programar en ROS, gracias a estos controladores se puede simular de manera real en Gazebo.

4.2.1.2. Modelado del sensor láser Hokuyo

De la misma manera que se crea el modelo del drone se debe crear el modelo 3D del sensor láser Hokuyo. La figura 4.12 muestra el modelo del sensor que se usará en la simulación. Del mismo modo, la programación que permite obtener la nube de puntos del láser se debe hacer en ROS.

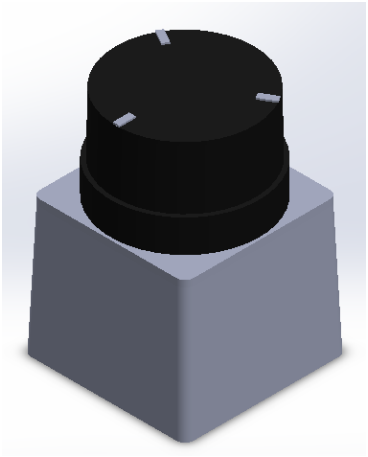


Figura 4.12: Modelo en 3D del sensor láser Hokuyo usando SolidWorks.

4.2.1.3. Modelo CAD a URDF

Gazebo utiliza un formato especial para poder simular modelos robóticos en 3D, este formato es conocido como **URDF** (Unified Robot Description Format). Debido a que en SolidWorks se obtienen modelos en formato CAD, Gazebo no puede "leer" directamente este tipo de formatos por lo que se debe pasar de formato CAD a URDF. El formato URDF tiene una estructura específica en la que se definen las distintas articulaciones y partes de un robot en las que se pueden especificar diversas características del robot, por ejemplo su

color, textura, tamaño, posición, etc. En la figura 4.13 se puede observar la estructura que tiene un archivo URDF.

```
<?xml version="1.0"?>

<robot name="Ardrone2.SLDASM">
  <link name="Cubierta_ardrone_base_link">
    <inertial>
      <origin
        xyz="0.0013233 -0.011773 0.011058"
        rpy="0 0 0" />
      <mass value="2.6891" />
      <inertia ixx="0.042687" ixy="1.3235E-05" ixz="-8.7629E-06" iyy="0.039095" iyz="0.0013657" izz="0.080131" />
    </inertial>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://Ardrone2.SLDASM/meshes/Cubierta_ardrone_base_link.STL" />
      </geometry>
      <material name="">
        <color rgba="0.79216 0.81961 0.93333 1" />
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://Ardrone2.SLDASM/meshes/Cubierta_ardrone_base_link.STL" />
      </geometry>
    </collision>
  </link>
</robot>
```

Figura 4.13: Estructura de un archivo URDF.

ROS incluye un paquete adicional que se instala junto con SolidWorks el cual permite exportar las partes y modelos de SolidWorks a un archivo URDF. Este paquete se puede obtener directamente de la página de ROS y bajo la denominación de ***sw_urdf_exporter***[13]. Se instala en el sistema operativo *Windows*.

En la capítulo de Manual de Usuario se explica cómo se obtiene el cambio de formato CAD a URDF usando esta herramienta y SolidWorks.

4.2.2. Localización y mapeado con Hector SLAM (sistemas de referencia)

El paquete de *hector_slam* (sección 4.1.4.4) contiene una serie de algoritmos que permiten simular en *Gazebo* un cuadricóptero que cuenta con una serie de sensores que son utilizados para la reconstrucción 2D del entorno de navegación en *Rviz*.

A la vez que se obtiene un mapa en 2D se consigue una estimación 3D del cuadricóptero mediante el uso de un paquete denominado *hector_pose_estimation*. Al crear un modelo en 3D del AR Drone 2.0 se puede implementar en la simulación de estos paquetes, pudiendo así simular cualquier plataforma robótica previamente creada y utilizando los algoritmos SLAM que el paquete de Hector proporciona. Los modelos, tanto robóticos como entornos, tienen que estar en el formato URDF para que puedan ser cargados en Gazebo.

Como se ha comentado anteriormente, las transformaciones de las coordenadas de los frames son importantes, por lo que para poder simular correctamente este paquete el árbol de transformaciones ha de ser el de la figura 4.14. Aquí se pueden observar las diferentes relaciones de transformación que se deben cumplir para que el sistema funcione correctamente. Si alguna está mal referenciada o no existe, lo más probable es que no

[13] http://wiki. ros.org/sw_urdf_exporter

se obtenga un resultado esperado o que los modelos 3D del cuadricóptero no se vean correctamente en Gazebo.

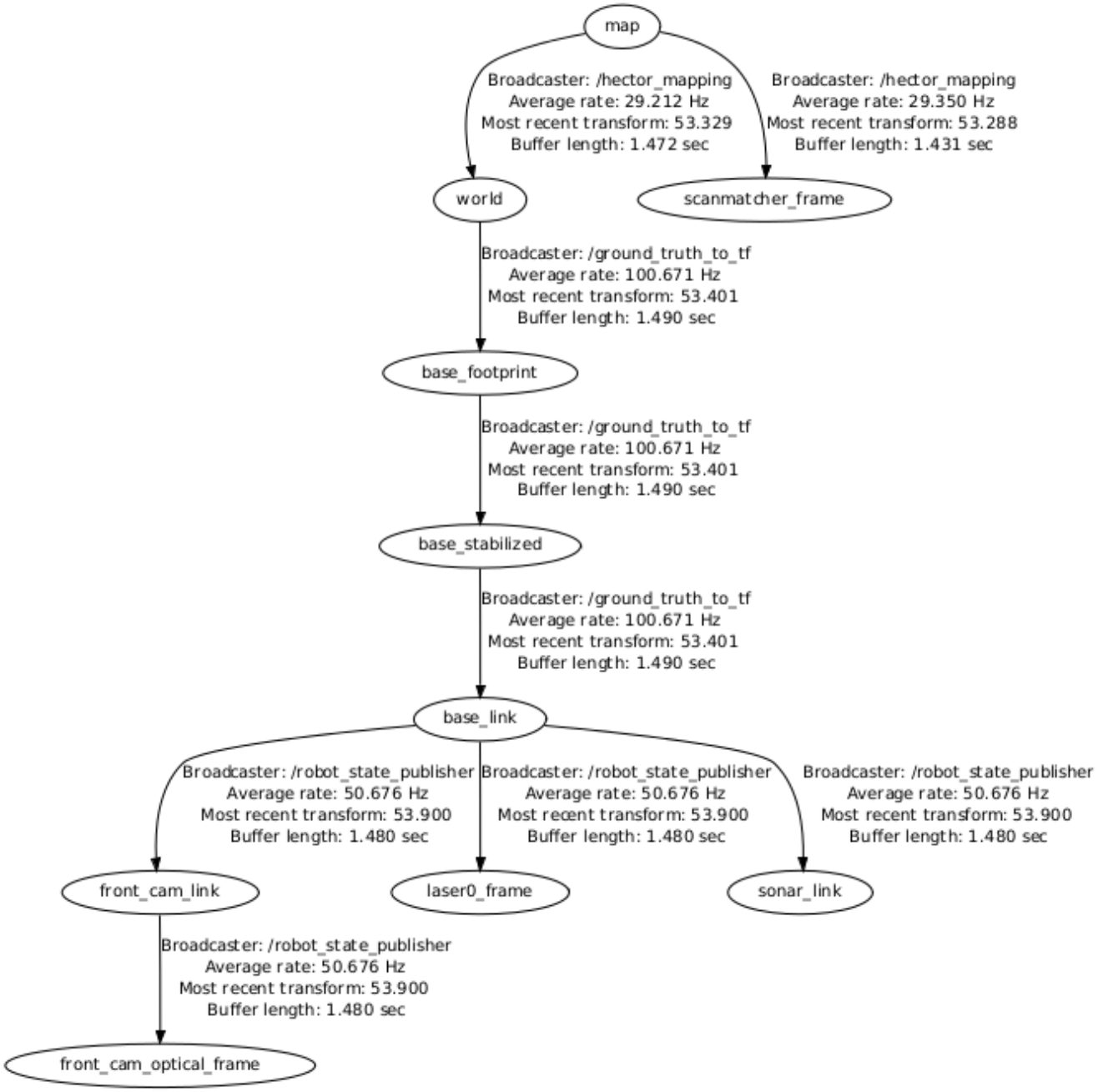


Figura 4.14: Árbol de transformaciones de simulación del cuadricóptero Hector.

En la siguiente imagen se puede ver el entorno de simulación *Gazebo* que cuenta con un entorno de oficinas y el modelo del cuadricóptero de Hector. Para el control del cuadricóptero es necesario la publicación de mensajes del tipo *geometry_msgs/Twist* en el topic *\cmd_vel* para así poder enviarle comandos de despegue, aterrizaje, giros y avance (los valores de avance por los ejes X, Y y Z están en metros mientras que las rotaciones están en radianes).

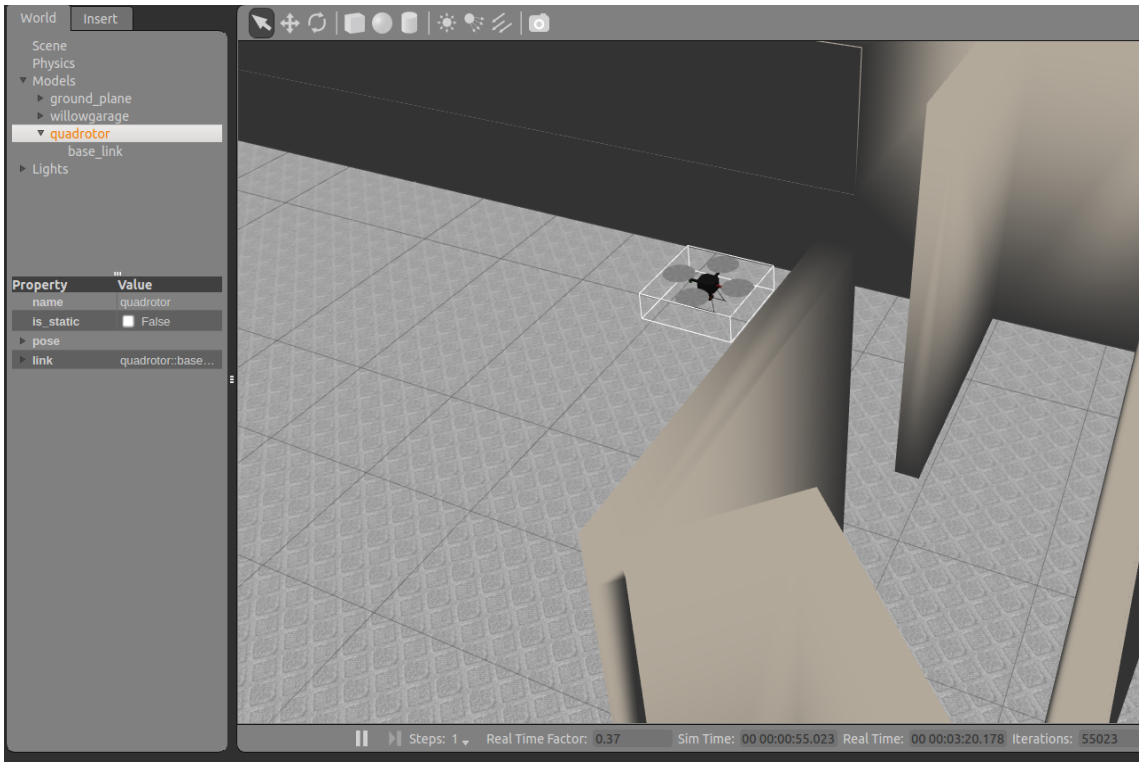


Figura 4.15: Simulación de cuadricóptero del paquete de Hector en Gazebo.

El resultado es el mismo si añadimos el modelo 3D creado:

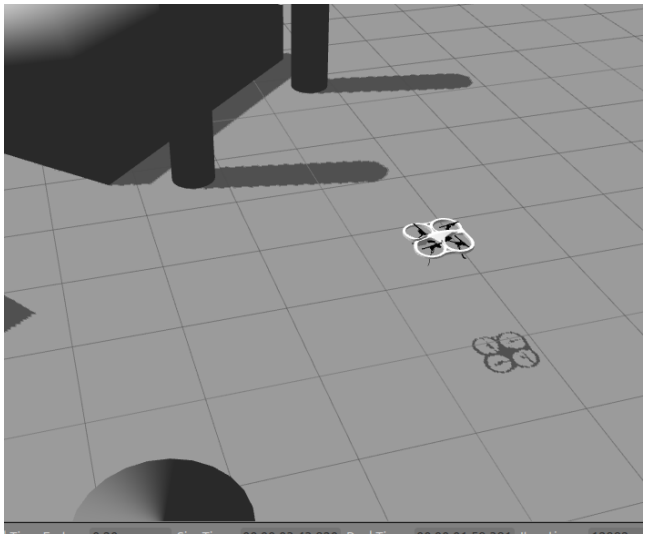


Figura 4.16: Simulación del AR Drone usando el paquete de Hector en Gazebo.

Así mismo, en Rviz se obtiene una representación del mapa en 2D y estimación de su posición en el mapa. .

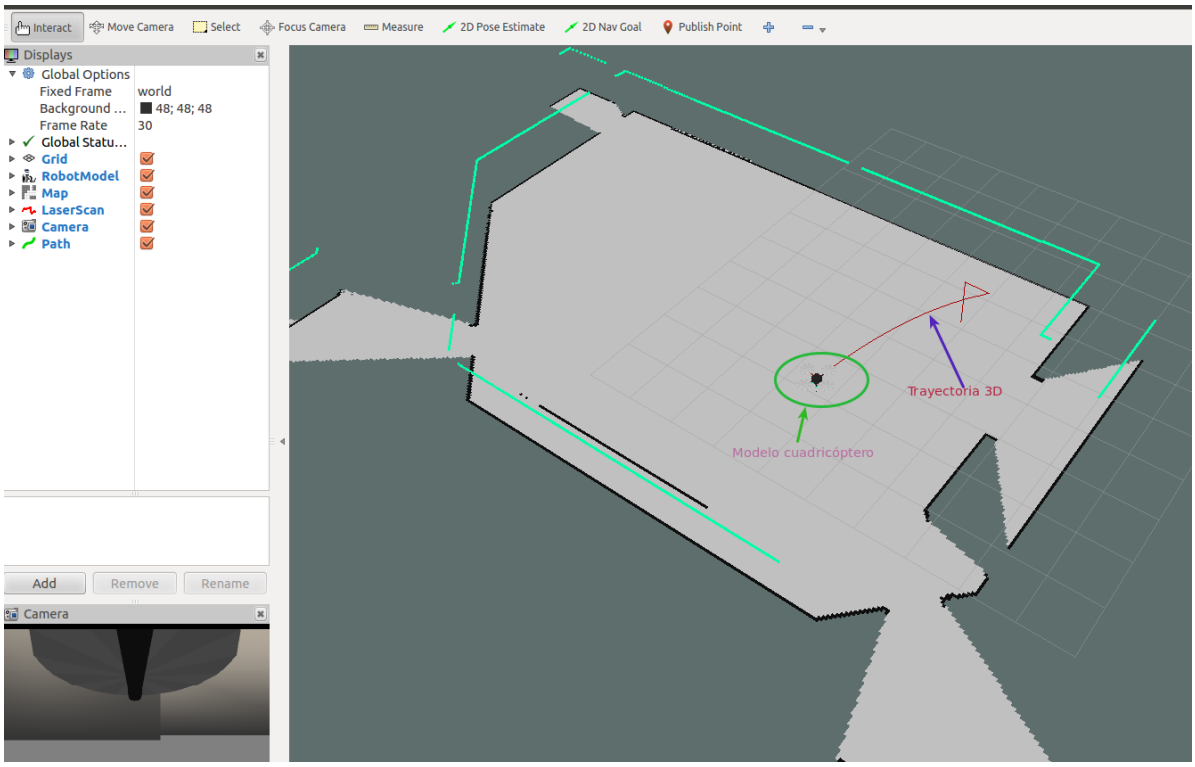


Figura 4.17: Recreación del mapa en 2D en Rviz con el paquete de Hector.

Hector cuenta con un estimador basado en un filtro de Kalman Extendido (EKF) para la estimación de posición 3D del robot. Esto se puede ver en la imagen 4.17. Para ello, utiliza un paquete denominado *hector_pose_estimation* que mediante la lectura de medidas de diferentes fuentes de sensores (barómetro, acelerómetros, ultrasonidos, sensores láseres, giróscopos, etc.) estima la posición 3D del robot en la obtención del mapa 2D.

4.3. Implementación sobre la plataforma real

En los apartados anteriores (ver sección 4.1 y siguientes) se han descrito los sensores necesarios para este proyecto. Diversas pruebas han sido necesarias con los sensores montados en el MAV para conseguir la mejor posición en éste. Se han conseguido mediante *acierto -error* y conocimientos básicos de física para intentar colocar lo más centrado posible el *centro de masas* (en el AR Drone 2.0) del conjunto "robot-sensores".

Algunas de las pruebas realizadas para esta parte del proyecto han sido colocar diversos trozos de madera de pesos conocidos y próximos a los pesos de los sensores colocados estratégicamente por la estrucutra del robot para conocer su peso máximo que puede aguantar y, con este peso, se han realizado pruebas de vuelo para ver si era lo suficientemente estable y así saber en qué posición deberían ir los sensores. Estas pruebas han sido útiles para observar que el peso total (robot-sensores) excede el peso admisible que el AR Drone 2.0 puede soportar para volar correctamente. Por ello, un objetivo principal en esta parte del proyecto ha sido conseguir reducir el peso para que el AR Drone pueda volar sin problemas. Algunos pasos que se han seguido para obtener este resultado han sido:

1. **Conocer el peso de cada sensor que se utilizarán.** Por un lado, el láser *Hokuyo* (sección 4.1.2) pesa 160 gramos, la *Raspberry Pi* unos 45 gramos, los sensores de

ultrasonidos 20 gramos y unas pequeñas patas que alargan las propias del drone para asegurar y proteger el sensor láser. El peso total que el AR Drone debe levantar, sin contar su peso, es de 225 gramos aproximadamente, a este peso hay que sumarle la carcasa de protección para interiores, por lo que el peso total aproximado de la plataforma completa es de **645 gramos**.

2. **Teleoperación.** El sensor láser se ha colocado en la base del AR Drone 2.0 y se ha modificado la carcasa para interiores para colocar la *Raspberry Pi* y a su vez se han quitado las pegatinas de decoración que incluye el AR Drone en su estructura para reducir el peso total lo máximo posible. Una vez montado, se han hecho las pruebas pertinentes y se ha estudiado su comportamiento para encontrar la mejor posición posible de la plataforma completa.
3. **Montar la estructura.**

Para establecer cuál era la mejor estructura para el montaje del drone, ha hecho falta la teleoperación del mismo. El control o teleoperación del AR Drone se ha realizado mediante teclado del PC a través de un algoritmo creado en ROS. Este control se realiza mediante el envío de mensajes sobre el topic *cmd_vel* (topic publicado por el paquete *adrone_autonomy*) del tipo *geometry_msgs* el cual permite el envío de comandos lineales y angulares para poder controlar el MAV desde teclado.

Antes de explicar cómo se ha implementado el algoritmo en el robot real, es necesario entender cómo funciona el sistema que se utiliza en este proyecto. En la figura 4.18 se pueden apreciar los diferentes módulos que forman el sistema completo:

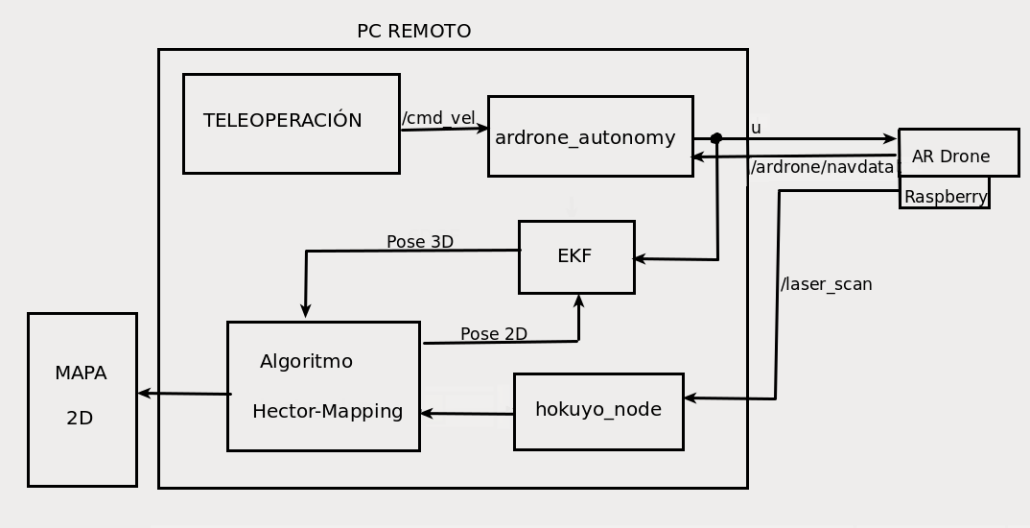


Figura 4.18: Sistema de bloques que explica los módulos utilizados en este proyecto.

Como se ha comentado anteriormente, la teleoperación se realiza mediante teclado a través del envío de los comandos de control al paquete *ardrone_autonomy* mediante el topic *\cmd_vel*. Una vez recibidos los comandos de control, el paquete envía las sentencias de ejecución al AR Drone y, a su vez, envía los comandos de entrada al estimador de posición 3D (módulo EKF en la figura) para que estime la posición del drone en el entorno y así, mediante el algoritmo Hector-Mapping, representar el mapa 2D e ir corrigiendo el mapa en función de la posición del drone. El algoritmo Hector-Mapping, además de utilizar los datos del EKF, necesita la nube de puntos del láser Hokuyo para la representación del

mapa y su corrección. Por ello, mediante el topic *laser_scan* la Raspberry envía los datos del láser acoplado al drone al algoritmo Hector-Mapping para obtener el mapa 2D.

Tras realizar los pasos anteriores la estructura final que se ha utilizado en el proyecto se puede observar en la figura 4.1.

4.3.1. Movimiento del robot por el entorno

Aunque se haya conseguido una estructura "estable", el controlador interno específico para su peso que lleva el AR Drone 2.0 ha hecho que sea difícil el control del mismo al añadirle un peso considerable. Por lo tanto, el AR Drone 2.0 no ha respondido de una manera óptima y se ha optado por realizar las pruebas del SLAM llevándolo a mano, no pudiéndose probar la parte del EKF, pero si el algoritmo Hector-Mapping.

Así, todas las pruebas y resultados obtenidos han sido con el MAV llevándolo a mano. Esto podría parecer un problema, pero al trabajar con el algoritmo Hector-Mapping (ver capítulo 3), al llevarlo a mano los movimientos son más estables para el láser, es decir, sin movimientos bruscos que puedan suponer un problema a la hora de realizar el SLAM y así se puedan obtener resultados mejores.

4.3.2. Mapeado 2D con Hector Mapping

Una vez claro cómo se moverá el drone por el entorno y haber realizado diferentes pruebas en simulación, el siguiente paso es realizar pruebas con el robot real. En este apartado se describirán los pasos necesarios para implementar correctamente el algoritmo Hector-Mapping en el AR Drone 2.0.

4.3.2.1. Corrección con los datos de la IMU

El AR Drone cuenta con una serie de sensores incorporados (ver 4.1.1) que le permiten navegar con una mayor estabilización. Al incorporar el sensor láser al AR Drone y el algoritmo Hector-Mapping, el drone irá estabilizando la visión del láser mediante la lectura de la IMU. Por ello, es necesario el uso de paquetes que permitan realizar una correcta estabilización del drone y alineación del láser con el suelo.

Una idea básica de lo que se pretende conseguir en este apartado viene reflejado en la imagen 4.19.

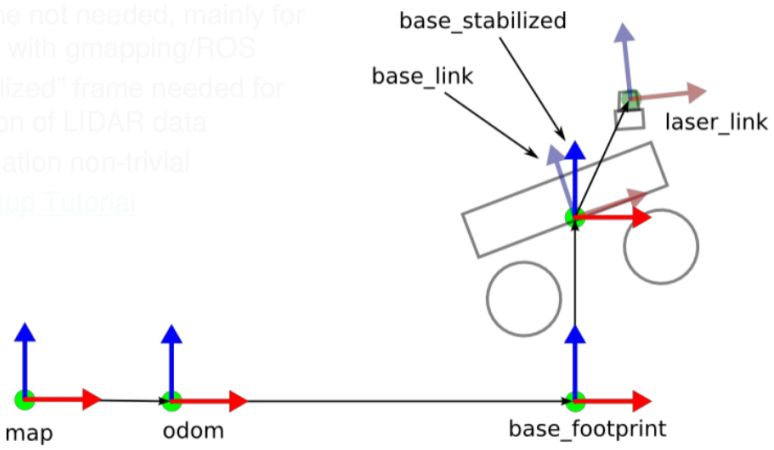


Figura 4.19: Transformaciones del sistema Hector Mapping.

En primer lugar es necesario tener claras las transformaciones. El paquete ardrone_autonomy publica un árbol de transformaciones entre estos frames: odom, *ardron_base_link*, *ardrone_base_frontcam* y *ardrone_base_bottomcam*.

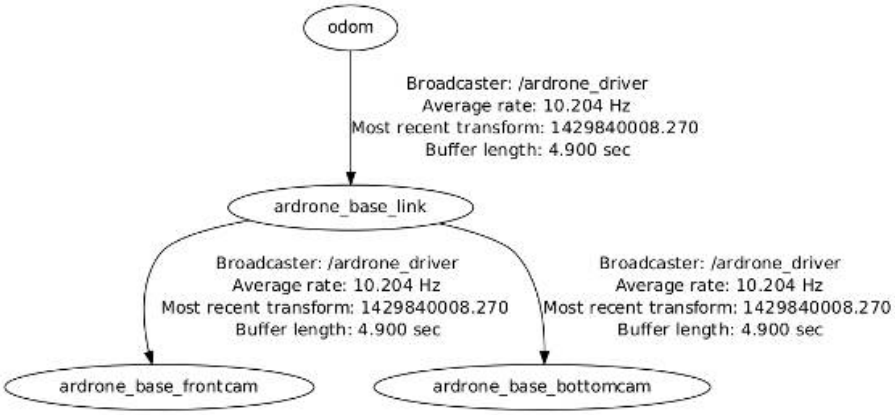


Figura 4.20: Árbol de transformaciones del AR Drone mediante el paquete ardrone_autonomy.

En la plataforma real, el frame *odom* es *base_stabilized* por lo que es necesario realizar una transformación de *odom* a *base_stabilized*. Como se puede ver en la imagen 4.19, el árbol de transformaciones ha de tener el siguiente aspecto:

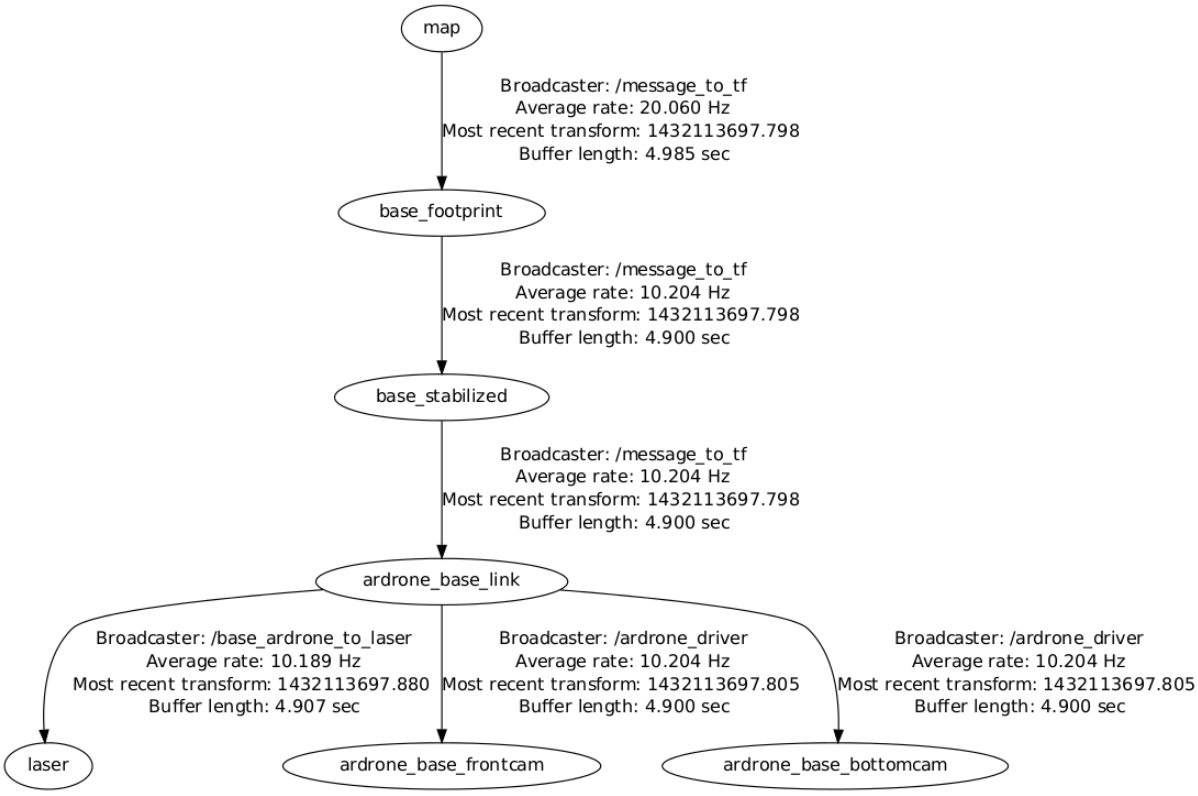


Figura 4.21: Árbol de transformaciones del AR Drone con el sistema de sensores completo.

Si observamos los árboles 4.21 y 4.14 son muy parecido ya que estamos utilizando el mismo algoritmo SLAM. La diferencia entre estos dos árboles, está en el publicador de transformaciones. Por un lado, en el árbol del robot real 4.21 está siendo publicada cada transformación por el paquete *message_to_tf* 4.1.4.4 el cuál permite la publicación de las transformaciones necesarias para la estabilización del AR Drone. Mientras que las trasnformaciones en simulación son publicadas mediante el algoritmo del Hector-Mapping (*ground_truth_to_tf*) que actúa de igual manera que el *message_to_tf* pero en simulación.

Para que el *message_to_tf* pueda realizar bien las publicaciones, necesita que la posición del robot se vaya publicando en un topic denominado *\pose* periódicamente y que existan los frames *map*, *base_footprint* y *base_stabilized*. Para que haya una relación entre el frame *base_footprint* y el *base_stabilized* se necesita utilizar el paquete *hector_imu_attitude_to_tf* ya que el paquete *message_to_tf* no publica la transformación los ángulos *roll* y el *pitch* que son necesarios para la estabilización del AR Drone y el láser Hokuyo.

Gracias al paquete *ardrone_autonomy* (sección 4.1.4), se puede enviar un comando de servicio el cual permite recalibrar las estimaciones de las rotaciones del AR Drone siempre que esté en una posición plana, es decir, en el suelo o en la plataforma de despegue. El servicio se envía con un mensaje vacío en el servicio "*Flattrim*":

```
$ rosservice call /ardrone/flattrim " "
```

Este servicio es necesario cada vez que se quiere volar el AR Drone ya que permite establecer unos valores iniciales de los sensores próximos a cero y así no se obtienen errores en el vuelo.

4.3.2.2. Pruebas de mapeado 2D

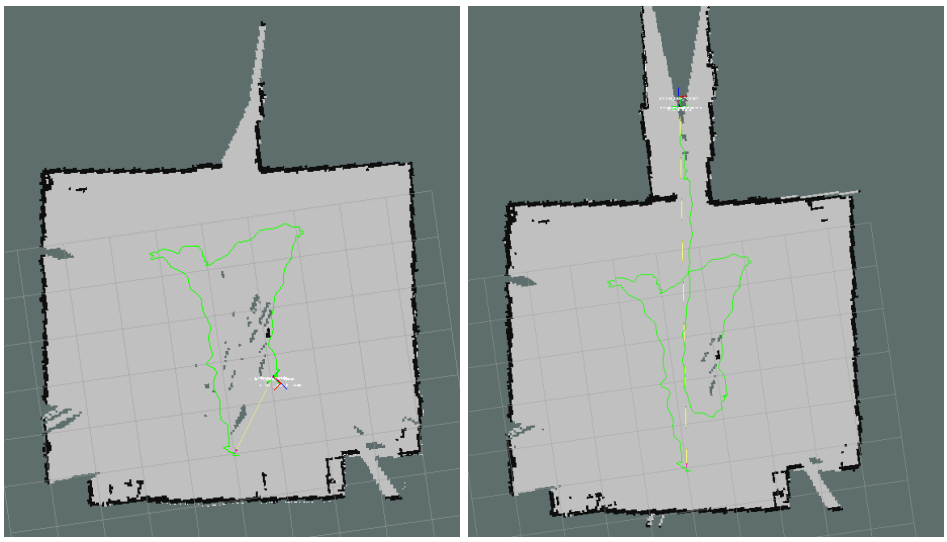
Una vez implementado todo el sistema del Hector-Mapping y comprobarlo que todo el sistema funciona correctamente en simulación, el siguiente paso es probar el robot con entornos reales.

Como se ha comentado anteriormente en el apartado 4.3.1 se ha decidido realizar las diferentes pruebas a mano.

Para comenzar es necesario asegurarse que todos los sensores, tanto el láser como los internos del AR Drone, funcionan y leen datos correctamente. El láser utiliza el paquete *hokuyo_node* y permite visualizar la nube de puntos del láser aunque el drone no esté volando, algo que es muy útil ya que se llevará a mano. Algunos de los sensores del AR Drone también se pueden comprobar, mediante el topic */adrone/navdata* del paquete *ardrone_autonomy* se puede ver cómo algunos de los valores de los sensores del drone cambian si movemos el AR Drone (aunque no este volando), como el sensor de altitud o estado de la batería entre otros.

Una vez comprobado que los sensores funcionan y que la comunicación con la Raspberry se realiza correctamente, se realizan algunas pruebas para comprobar que el sistema SLAM funciona correctamente.

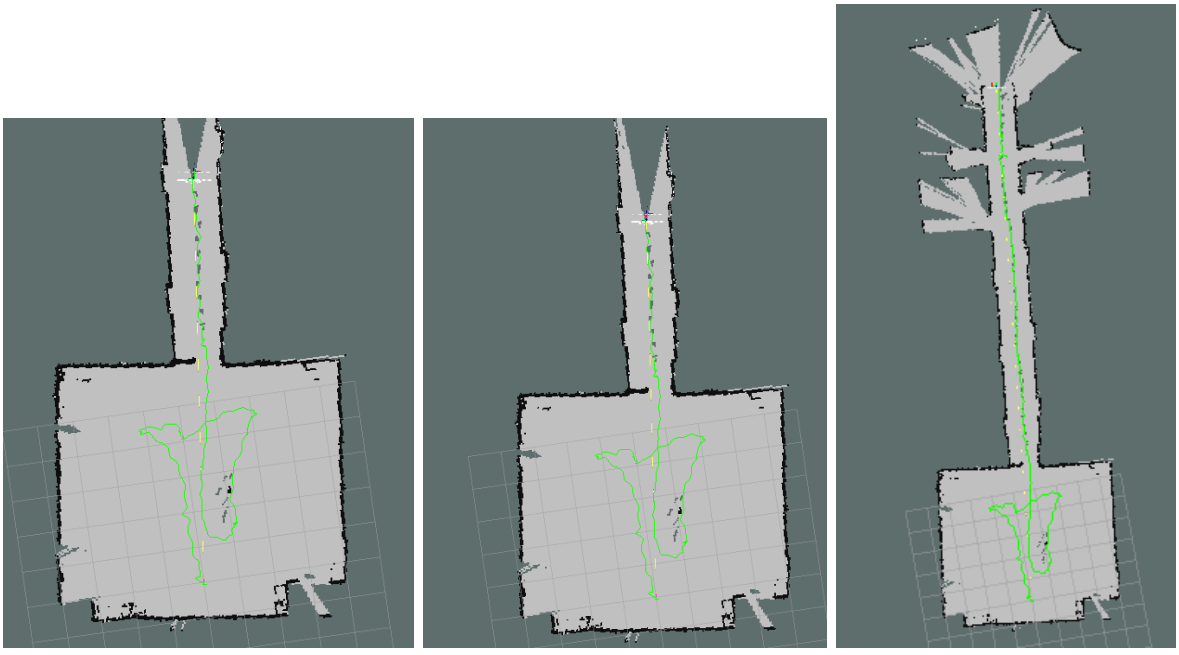
Las siguientes imágenes muestran cómo se obtiene el mapa en 2D con el algoritmo Hector-Mapping en Rviz.



(a) (b)

Figura 4.22: Mapa 2D con algoritmo Hector-Mapping en plataforma real. El robot recorre una habitación y un pasillo del Departamento de Electrónica.

Como se puede observar en las imágenes 4.22 el algoritmo implementado en la plataforma real funciona. Se puede comprobar que la figura 4.17 (mapa 2D en simulación) concuerda con el resultado obtenido con la plataforma real.



(a) (b) (c)

Figura 4.23: Mapa 2D con algoritmo Hector-Mapping en plataforma real. Pasillos del Departamento de Electrónica, en las imágenes se pueden observar el pasillo y los despachos.

Algunas de las características negativas del algoritmo Hector-Mapping es que no implementa un sistema de lazo cerrado y que en entornos amplios y con pocas características

no funciona tan bien como en pequeños ya que va perdiendo información de odometría, y al no tener un sistema de lazo cerrado, se obtienen resultados confusos. Esto se puede comprobar en las imágenes de la figura 4.23 en las que se aprecia cómo a medida que se va avanzando por el entorno, la representación del mapa empieza a "inclinarse" en el plano sin llegar a ser totalmente paralela y perpendicular al muro donde comenzó la representación 2D.

En las imágenes anteriores 4.23 se puede ver la trayectoria 2D que ha realizado el drone, esto es, por el uso del paquete *hector_localization*. Aunque estima la posición 2D, este paquete debería estimar la trayectoria en 3D del robot, pero no se ha conseguido implementar este paquete al proyecto. Por ello, en el siguiente apartado se explica una solución alternativa que se ha realizado para el estimar la posición 3D del robot.

4.3.3. Estimación de la posición 3D mediante filtro de Kalman extendido

El algoritmo *Hector-Mapping* incluye un estimador de posición 3D, pero como se ha comentado en el capítulo 3, no se ha podido utilizar este estimador en este proyecto por lo que se ha optado crear un algoritmo de estimación de posición basándose en el filtro de Kalman Extendido (sección 2.3.1.2). La obtención de estimación de posición 3D sin odometría es uno de los desafíos más importantes del SLAM [14], por ello es necesario la creación de este algoritmo EKF, el cual se ha realizado en MATLAB[14]. Esta parte no se ha podido probar en la plataforma por diversos motivos, uno de ellos fue que se dedicó mucho más tiempo a los otros estimadores sin obtener resultados buenos, otro motivo es que trabaja en paralelo al mapa 2D, es decir, estima la posición del AR Drone en el entorno MATLAB sin dibujar la trayectoria del drone en el mapa obtenido y se realiza de manera "*off-line*" teniendo que guardar las lecturas de los sensores del drone previamente para después probarlos en el estimador 3D.

Con el fin de fusionar todos los datos disponibles, se emplea un filtro de Kalman extendido (EKF). Por ello, en los siguientes subapartados se describirán los módulos del sistema SLAM mostrado en la figura 4.24.

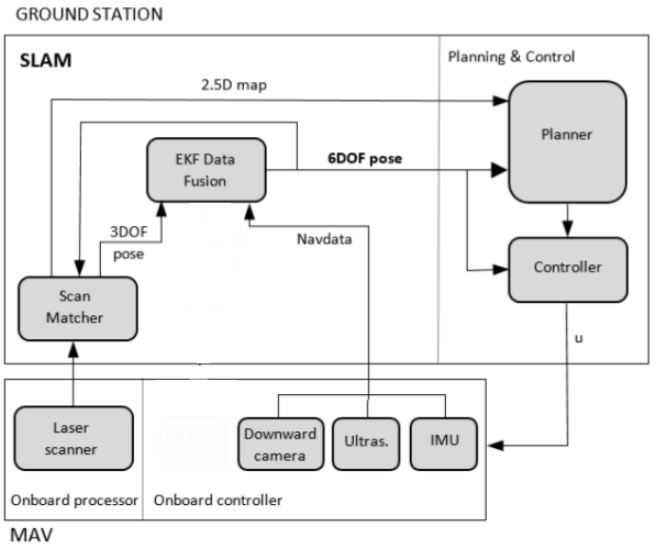


Figura 4.24: Diagrama de bloques de la plataforma real.

[14]Es una herramienta de software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M).

A continuación se describirá el sistema EKF.

1. **Vector de estado para el EKF.**

$$X_t = (x_t, y_t, z_t, vx_t, vy_t, vz_t, \Phi_t, \Theta_t, \Psi_t, \dot{\Psi}_t)^T \quad \in \quad \Re^{10} \tag{4.1}$$

Donde (x_t, y_t, z_t) es la posición de la MAV en m, (vx_t, vy_t, vz_t) la velocidad en m/s, $(\Phi_t, \Theta_t, \Psi_t)$ los ángulos roll, pitch y yaw en grados y $\dot{\Psi}_t$ la velocidad de yaw-rotación en $grados/s$, todas ellas en las coordenadas del mundo.

2. **Entradas y salidas del filtro EKF.** La señal de control con la que se actuará sobre el drone es:

$$u = (\hat{\Phi}, \hat{\Theta}, \hat{\dot{\Psi}}, \hat{vz})^T \tag{4.2}$$

Este EKF tiene dos tipos de observaciones o medidas:

- Medidas que se obtienen del AR Drone (mediante el uso del paquete NAVDATA del *ardrone_autonomy*):

$$z_{\mathbf{NAVDATA}} = (\bar{vdx}, \bar{vdy}, \bar{z}, \bar{\Phi}, \bar{\Theta}, \bar{\Psi})^T \tag{4.3}$$

Donde $\bar{vdx}$ y $\bar{vdy}$ son las medidas de las velocidades horizontales del drone en su sistema de coordenadas local que se transforman a partir de las coordenadas del mundo vx_t y vy_t, $\bar{z}$ es la altura medida y $\bar{\Phi}$, $\bar{\Theta}$ y $\bar{\Psi}$ son los ángulos medidos del acelerómetro.

- La observación de posición 2D obtenida con el Hector-Mapping a partir del proceso *Scan-Matching*:

$$z_{\mathbf{LASER}} = (x_L, y_L, \Psi_L)^T \tag{4.4}$$

Donde x_L y y_L son las coordenadas de la nube de puntos en el mundo y Ψ_L es el ángulo de orientación del láser.

A continuación, se definen los modelos de predicción y observación.

3. **Modelos.**

a) **Modelo de predicción**: El modelo de predicción se basa en el modelo de movimiento completo de la dinámica de vuelo del MAV [15]. Se ha realizado una nueva calibración de los parámetros del modelo ya que el peso del sensor láser Hokuyo modifica considerablemente la dinámica del sistema. El modelo establece que la aceleración horizontal del MAV es proporcional a la fuerza horizontal que actúa sobre él, es decir, la fuerza de aceleración menos la fuerza de arrastre. El arrastre es proporcional a la velocidad horizontal del MAV, mientras que la fuerza de aceleración es proporcional a la proyección del eje Z en el plano horizontal, lo que conduce a:

$$vx_t = K_1(K_2(cos(\Psi_t)\ sin(\Phi_t)\ cos(\Theta_t)\ - sin(\Psi_t)\ sin(\Theta_t)\ ) - vx_t) \tag{4.5}$$

$$vy_t = K_1(K_2(-sin(\Psi_t)\ sin(\Phi_t)\ cos(\Theta_t)\ - cos(\Psi_t)\ sin(\Theta_t)\ ) - vy_t) \tag{4.6}$$

Por otra parte, la influencia de la orden de control enviada $u = (\hat{\Phi}, \hat{\Theta}, \hat{\Psi}, \hat{vz})$ se

describe por el siguiente modelo lineal:

$$\dot{\Phi}_t = K_3(K_4\hat{\Phi}_t - \Phi_t) \tag{4.7}$$

$$\dot{\Theta}_t = K_3(K_4\hat{\Theta}_t - \Theta_t) \tag{4.8}$$

$$\ddot{\Psi}_t = K_5(K_6\hat{\dot{\Psi}}_t - \dot{\Psi}_t) \tag{4.9}$$

$$vz_t = K_7(K_8\hat{vz}_t - vz_t) \tag{4.10}$$

Reuniendo las ecuaciones en un formato apropiado, el modelo de predicción (no lineal) $\chi_t = g(\chi_{t-1}, u_t)$ queda:

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \\ z_{t+1} \\ vx_{t+1} \\ vy_{t+1} \\ vz_{t+1} \\ \Phi_{t+1} \\ \Theta_{t+1} \\ \Psi_{t+1} \\ \dot{\Psi}_{t+1} \end{pmatrix} \leftarrow \begin{pmatrix} x_t \\ y_t \\ z_t \\ vx_t \\ vy_t \\ vz_t \\ \Phi_t \\ \Theta_t \\ \Psi_t \\ \dot{\Psi}_t \end{pmatrix} + \Delta_t \cdot \begin{pmatrix} vx_t \\ vy_t \\ vz_t \\ K_1(K_2(cos(\Psi_t)sin(\Phi_t)cos(\Theta_t) - sin(\Psi_t)sin(\Theta_t)) - vx_t) \\ K_1(K_2(-sin(\Psi_t)sin(\Phi_t)cos(\Theta_t) - cos(\Psi_t)sin(\Theta_t)) - vy_t) \\ K_3(K_4\hat{\Phi}_t - \Phi_t) \\ K_3(K_4\hat{\Theta}_t - \Theta_t) \\ K_5(K_6\hat{\dot{\Psi}}_t - \dot{\Psi}_t) \\ \dot{\Psi}_t \\ K_7(K_8\hat{vz}_t - vz_t) \end{pmatrix} \tag{4.11}$$

Las constantes K_i se obtienen de manera experimental mediante el uso de los datos registrados y obtenidos del propio AR Drone. En la sección 4.3.3.1 se describe cómo se han obtenido dichas constantes.

b) **Modelo de observación o corrección del NAVDATA**: Este modelo relaciona las mediciones a bordo obtenidas a través del canal de navegación del drone (Navdata).

$$z_{NAVDATA} = h_{NAVDATA}(\chi_t) \tag{4.12}$$

$$\mathrm{h}_{NAVDATA}(\chi_t) = \begin{pmatrix} vx_t \cdot cos(\Psi_t - vy_t \cdot sin(\dot{\Psi}_t)) \\ vx_t \cdot sin(\Psi_t - vy_t \cdot cos(\dot{\Psi}_t)) \\ vz_t \\ \Phi_t \\ \Theta_t \\ \dot{\Psi}_t \end{pmatrix} \tag{4.13}$$

c) **Modelo de observación o corrección del SCANMATCHER**: a diferencia de los anteriores modelos, este modelo es lineal:

$$H_{LASER} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.14}$$

Tras explicar las ecuaciones que se utilizarán en el EKF, se pasará a la descripción de la obtención de las constantes y de la programación del filtro.

4.3.3.1. Identificación experimental de los parámetros del modelo de predicción

En el modelo de predicción del EKF aparecen 8 parámetros $K_{1...8}$, de los cuales se ha realizado una identificación experimental, de manera que se ajusten al comportamiento real del ARDrone.

A continuación se muestra el procedimiento realizado para dicha identificación.

a) **Identificación de K_7 y K_8.** Estos parámetros aparecen en la siguiente ecuación de estado:

$$\dot{vz}_t = K_7(K_8 \cdot \hat{vz}_t - vz_t) \tag{4.15}$$

Esta ecuación caracteriza la evolución de la velocidad vertical del Drone vz ante la aplicación de un comando de velocidad vertical . Si se expresa dicha ecuación en el dominio de Laplace, se obtiene:

$$\frac{Vz(s)}{\hat{Vz}(s)} = \frac{K_8}{K_7 \cdot s + 1} \tag{4.16}$$

Por lo que el parámetro K_8 representa la ganancia estática entre el comando de velocidad vertical enviado (valor normalizado entre ± 1) y la velocidad vertical realmente alcanzada, y el parámetro K_7 corresponde a la inversa de la constante de tiempo de dicha evolución.

Para obtener estos parámetros, se ha enviado al AR Drone el comando $u = [0001]$, que corresponde a un escalón máximo en la velocidad vertical, y se han registrado los datos de altura obtenidos por el ultrasonido vertical. Derivando dicha altura se obtiene la velocidad vertical mostrada en la siguiente figura 4.25, que corresponde al siguiente valor aproximado para los parámetros $K_7 = 5$ y $K_8 = 0{,}8$:

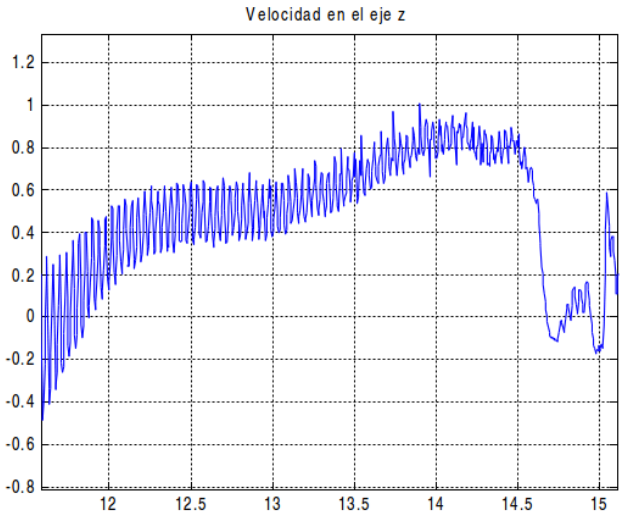


Figura 4.25: Identificación de $K_7 = 5$ y $K_8 = 0{,}8$.

b) **Identificación de K_5 y K_6.** Estos parámetros aparecen en la siguiente ecuación de estado:

$$\ddot{\Psi}_t = K_5 \cdot (K_6 \cdot \hat{\dot{\Psi}}_t) \tag{4.17}$$

Esta ecuación caracteriza la evolución de la velocidad de giro del Drone entorno al eje z ante la aplicación de un comando de giro. Igual que en el caso anterior, K_6 representa la ganancia estática y K_5 la inversa de la constante de tiempo.

Para obtener estos parámetros, se ha enviado al AR Drone el comando $u = [0010]$, que corresponde a un escalón máximo en la velocidad de giro entorno al eje z, y se han registrado los datos del ángulo "yaw", que posteriormente se han derivado para obtener la velocidad. Los datos obtenidos se muestran en la siguiente gráfica 4.26, de la que se obtienen los siguiente parámetros aproximados $K_5 = 1$ y $K_6 = 10$:

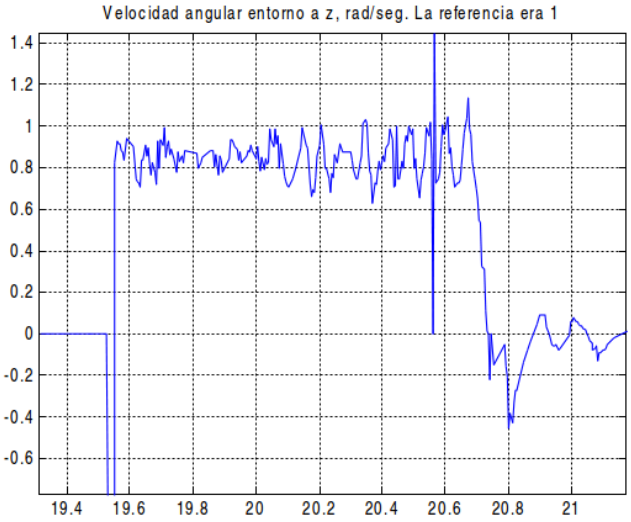


Figura 4.26: Identificación de $K_5 = 1$ y $K_6 = 10$.

c) **Identificación de K_3 y K_4.** Estos parámetros aparecen en la siguiente ecuación de estado:

$$\dot{\Phi}_t = K_3 \cdot (K_4 \cdot \hat{\Phi}_t - \Phi_t) \tag{4.18}$$

$$\dot{\Theta}_t = K_3 \cdot (K_4 \cdot \hat{\Theta}_t - \Theta_t) \tag{4.19}$$

En las que se supone que el movimiento lateral del drone es igual a lo largo del eje x al modificar el pitch, que a lo largo del eje y al modificar el roll. Efectivamente, de manera experimental se obtienen valores muy similares, por lo que se muestra a continuación sólo la identificación realizada con la modificación del roll. Igual que en los casos anteriores, se envía un comando escalón, en este caso $u = [1000]$, y se registran los datos del ángulo roll obtenidos de los giróscopos del AR Drone, permitiendo esto obtener la siguiente gráfica 4.27, y el siguiente valor aproximado para los parámetros $K_3 = 1{,}66$ y $K_4 = 0{,}6$:

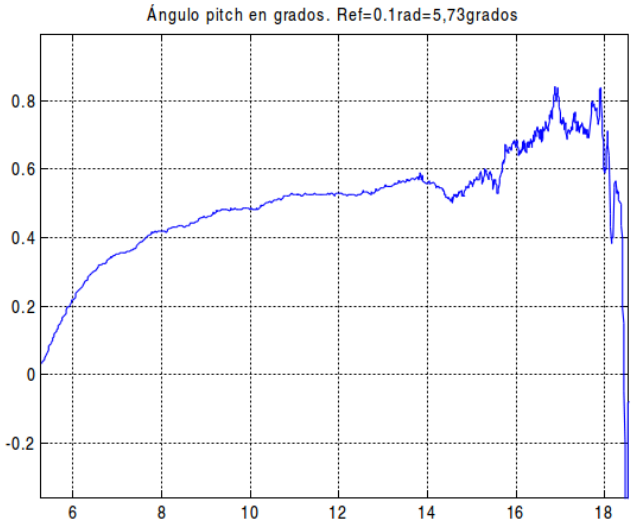


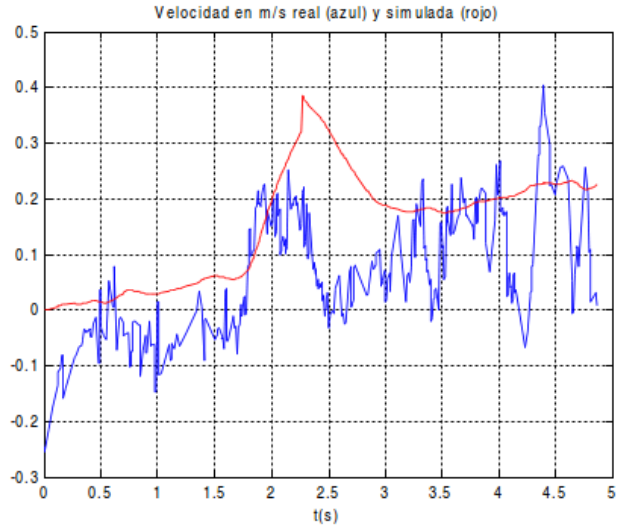
Figura 4.27: Identificación de $K_3 = 1{,}66$ y $K_4 = 0{,}6$.

d) **Identificación de K_1 y K_2.** Las ecuaciones para la identificación de K_1 y K_2 son:

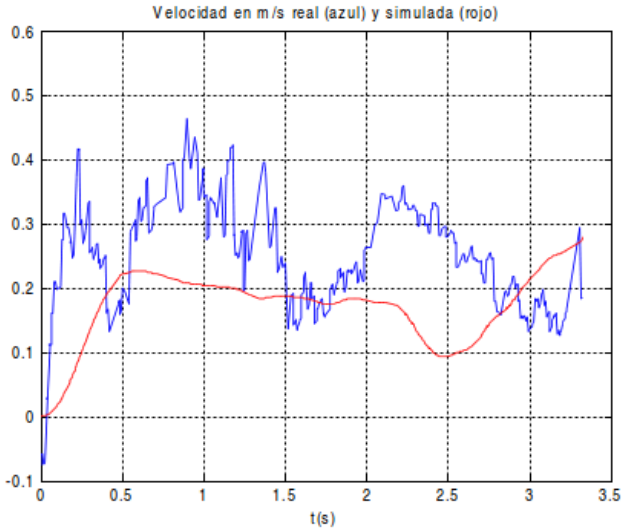
$$\dot{vx}_t = K_1 \cdot (K_2 \cdot (cos(\Psi_t) \cdot sin(\Phi_t) \cdot cos(\Theta_t) - sin(\Psi_t) \cdot sin(\Theta_t)) - vx_t) \quad (4.20)$$

$$\dot{vy}_t = K_1 \cdot (K_2 \cdot (-sin(\Psi_t) \cdot sin(\Phi_t) \cdot cos(\Theta_t) - cos(\Psi_t) \cdot sin(\Theta_t)) - vy_t) \quad (4.21)$$

Estas ecuaciones de estado son no lineales y por lo tanto la identificación resulta más compleja, por lo que se ha realizado mediante un proceso de prueba y error, simulando la salida de dichos modelos ante una entrada determinada, y comparándola con la salida real obtenida al enviar esa misma entrada al AR Drone. Se han realizado dos experimentos para este fin, uno de ellos enviándole al Ardrone un comando de roll (velocidad en el eje y) y otro enviando un comando de pitch (velocidad en el eje x). Tras el ajuste experimental se llega a las siguientes gráficas 4.28, que comparan en cada uno de los dos casos, las velocidades reales y las obtenidas por el modelo.



(a)



(b)

Figura 4.28: Identificación de K_1 y K_2

Estas gráficas se han obtenido para unos valores de K1=10 y K2=3, para los cuales se ha obtenido la mejor aproximación posible, a pesar del alto nivel de ruido que tienen las señales.

4.3.3.2. Programación del filtro de Kalman Extendido

En los siguientes apartados se explica la programación del EKF y los diferentes resultados obtenidos.

Etapa de predicción.

La etapa de predicción devuelve un nuevo estado y una nueva covarianza a partir del estado anterior, la covarianza P y la señal de control.
Las ecuaciones en esta etapa son:

$$\bar{\chi}_t = g(\chi_{t-1}, u_t) \tag{4.22}$$

$$\bar{P}_t = G_t \cdot P_{t-1} \cdot G_t^T + V_t \cdot M_t \cdot V_t^T \tag{4.23}$$

Donde G es la jacobiana de g respecto al estado χ y V es la jacobiana de g respecto a la señal de control u.

Corrección con las medidas del NAVDATA

La etapa de corrección devuelve un nuevo estado y una nueva covarianza (menor a la anterior) a partir del estado anterior, la covarianza P y las medidas del NAVDATA. Las ecuaciones en esta etapa son:

$$K_t = \bar{P}_t H_t^T (H_T \bar{P}_t H_t^T + Q_t)^{-1} \tag{4.24}$$

$$\chi_t = \bar{\chi}_t + K_t(z_t - h(\bar{\chi}_t)) \tag{4.25}$$

$$\bar{P}_t = (I - K_t H_t)\bar{P}_t \tag{4.26}$$

Donde H es la jacobiana del modelo de observación respecto al estado χ.

Corrección con las medidas del láser

Recibe el estado anterior, la covarianza P anterior y las medidas del láser. Devuelve el nuevo estado, y la nueva covarianza (siempre menor que la anterior).

Como este modelo sí que es lineal, se puede utilizar las ecuaciones propias del KF. Por lo tanto, en este caso H es directamente el modelo lineal de observación, en lugar de su jacobiana.

Las ecuaciones son las mismas que las ecuaciones 4.24, 4.25 y 4.26, teniendo en cuenta que H es un modelo lineal.

4.3.3.3. Pruebas y resultados

Se ha programado en Matlab un simulador del comportamiento del robot, para poder validar el funcionamiento del EKF en diferentes condiciones.

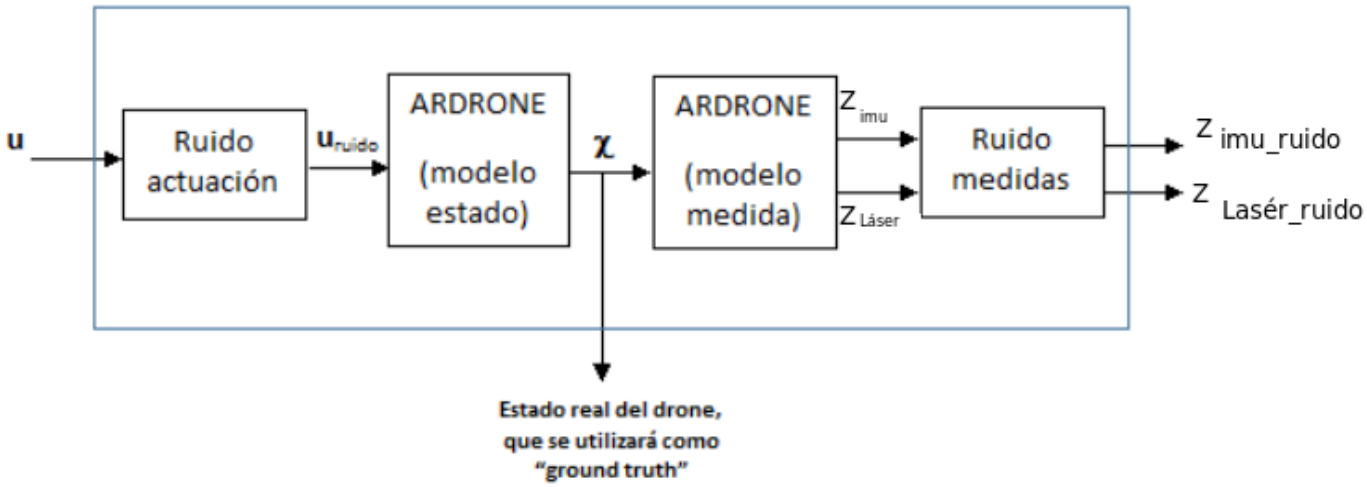


Figura 4.29: Diagrama de bloques que explica cómo funciona el filtro de Kalman Extendido.

Las diferentes pruebas realizadas muestran cómo actúa el EKF creado. Cada prueba tiene una configuración distinta por lo que hay que modificar los ficheros del algoritmo en MATLAB con los parámetros de cada configuración.
A continuación se muestran las diferentes pruebas con la configuración mostrada en la figura 4.30.

Tipos de Ruido	Valores
Actuación	**0.3** sobre todas las señales normalizadas. Equivale a un 30 % de la velocidad máx.
Láser	0.2 metros para longitud y 0.2 radianes para ángulo
IMU	Vector de estados inicial de la IMU: [0,1 0,1 0,2 0,1 0,1 0,2]

Tabla 4.4: Configuración de parámetros para realizar prueba con el EKF.

Configuración 1: Avance en el eje Z.

Tras explicar qué valores son necesarios para esta prueba de ascenso por el eje Z, se ha configurado las siguientes matrices de covarianza para el ruido en el EKF:

```
%Covarianzas de los errores
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Mt=0.4*eye(4,4);  %Covarianza ruido actuación. Supongo que es sobre los comandos normalizados [-1,1]

Qt_laser=[0.2 0 0;0 0.2 0;0 0 0.2]; %Covarianza ruido medida del laser (en m,m,rad)
Qt_imu=zeros(6,6);%Covarianza ruido medida de la imu
Qt_imu(1,1)=0.1;  %velocidad x en m/s
Qt_imu(2,2)=0.1;  %velocidad y en m/s
Qt_imu(3,3)=0.2;  %altura en m
Qt_imu(4,4)=0.1; %roll en rad
Qt_imu(5,5)=0.1; %pitch en rad
Qt_imu(6,6)=0.2;  %yaw en rad


%Inicialización del estado y su covarianza
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
estado(1,:)=[0 0 0 0 0 0 0 0 0 0];
covarianza(:,:,1)=[0.1 0 0 0 0 0 0 0 0 0;
                   0 0.1 0 0 0 0 0 0 0 0;
                   0 0 0.1 0 0 0 0 0 0 0;
                   0 0 0 0.1 0 0 0 0 0 0;
                   0 0 0 0 0.1 0 0 0 0 0;
                   0 0 0 0 0 0.1 0 0 0 0;
                   0 0 0 0 0 0 0.03 0 0 0;
                   0 0 0 0 0 0 0 0.03 0 0;
                   0 0 0 0 0 0 0 0 0.08 0;
                   0 0 0 0 0 0 0 0 0 0.08];
```

Figura 4.30: Prueba EKF: Configuración 1.

Una vez configurada todas las matrices de covarianza, se introduce una señal de control para que se eleve el drone (drone simulado en MATLAB) por el eje Z, siendo la señal de control $u = [0\ 0\ 0\ 1]'$. Se han realizado diferentes pruebas utilizando por un lado la etapa de predicción y por otro la etapa sin predicción, en color rojo se mostrará la trayectoria real del drone y en azul la estimación de posición 3D.
Las siguientes imágenes muestran los resultados obtenidos de las diferentes pruebas realizadas sin el filtro de Predicción.

- **Etapa sin Predicción del filtro: Sólo medidas del láser.** Esta prueba se ha realizado sin el filtro de predicción y utilizando sólo las medidas del láser. En la siguiente imagen se puede observar que, al estar utilizando sólo las medidas del láser y sin el filtro de predicción, la estimación de posición se hace en el plano 2D del mapa.

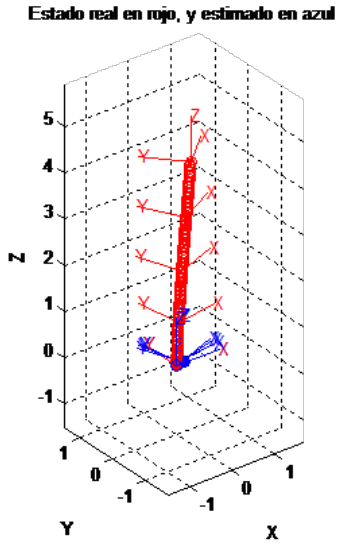


Figura 4.31: Etapa de Predicción del filtro prueba 1: Sólo medidas del láser.

- **Etapa sin Predicción del filtro: Sólo medidas de la IMU.** Para esta prueba, se mantiene la configuración anterior pero esta vez se usa sólo las medidas de la IMU. En ella se obtienen mejores resultados que la prueba anterior ya que las medidas de la IMU se obtienen en 3D (x, y, z) y el láser sólo en 2D (ejes x e y) como se puede observar en las imágenes 4.31 y 4.32. Sin embargo, al usar la etapa sin el filtro de Predicción la estimación de posición no es la deseada.

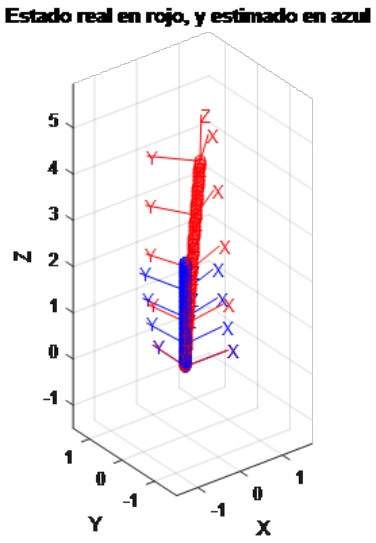


Figura 4.32: Etapa de Predicción del filtro prueba 1: Sólo medidas de la IMU.

- **Etapa sin Predicción del filtro: Medidas de la IMU y el Láser.** Esta prueba se realiza con las medidas de la IMU y el láser. En la siguiente imagen se puede observar que al usar tanto las medidas de la IMU como las del láser, la trayectoria estimada es mejor que las pruebas anteriores y es debido a que se obtienen mejores resultados combinando estas medidas, aunque el resultado sigan siendo los no deseados ya que no se usa la etapa de Predicción.

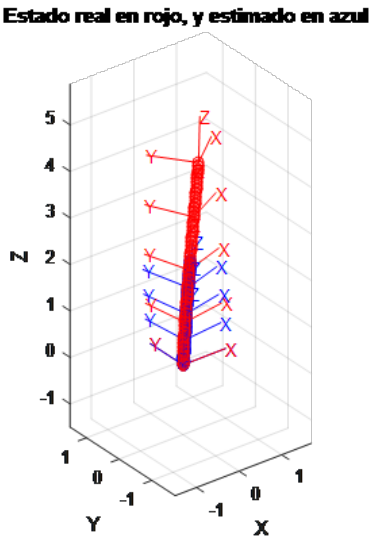


Figura 4.33: Etapa sin Predicción del filtro prueba 1: Medidas de la IMU y el Láser.

Las siguientes pruebas se realizan con la etapa de Predicción.

- **Etapa de Predicción del filtro: Sin medidas.** En esta prueba se usa sólo la etapa de predicción sin medidas. En la siguiente imagen se puede observar que aunque no se usen medidas, la etapa de Predicción mejora la estimación de la posición del drone aproximándose a la trayectoria real.

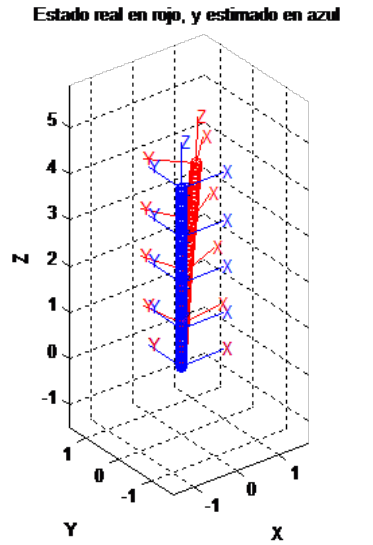


Figura 4.34: Etapa de Predicción del filtro prueba 1: Sin medidas.

- **Etapa de Predicción del filtro: Sólo medidas del láser.** La siguiente imagen muestra la estimación de posición 3D del drone con la etapa de Predicción y usando las medidas del láser. Se puede observar que usando el láser se mejora mucho la estimación, aunque la estimación no llegue a la posición final del drone.

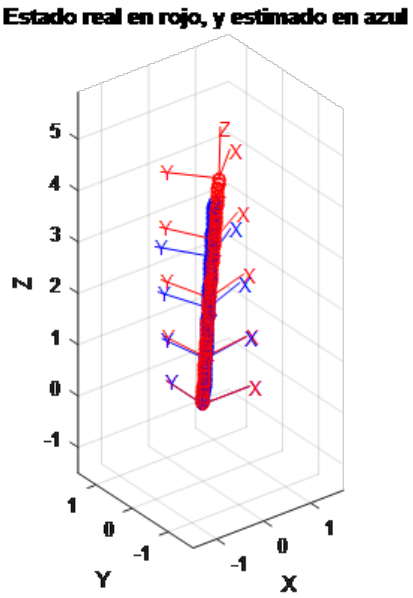


Figura 4.35: Etapa de Predicción del filtro prueba 1: Sólo medidas del láser.

- **Etapa de Predicción del filtro: Sólo medidas de la IMU.** Para esta prueba, se utilizan las medidas de la IMU. En la siguiente imagen se puede comprobar que la estimación de predicción se aproxima bastante a la deseada aunque no se use las medidas del láser.

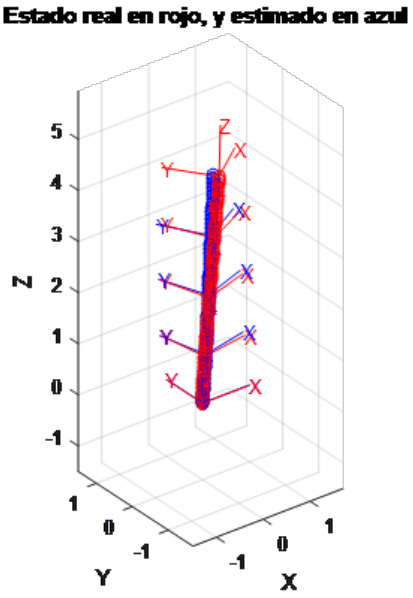


Figura 4.36: Etapa de Predicción del filtro prueba 1: Sólo medidas de la IMU.

- **Etapa de Predicción del filtro: Medidas de la IMU y el Láser.** Por último, se realiza una prueba en la que se utilizan las medidas de la IMU y el láser. En la imagen 4.37 se puede observar que el uso de la etapa de Predicción en conjunto con las medidas de la IMU y el láser estiman correctamente la posición 3D del drone.

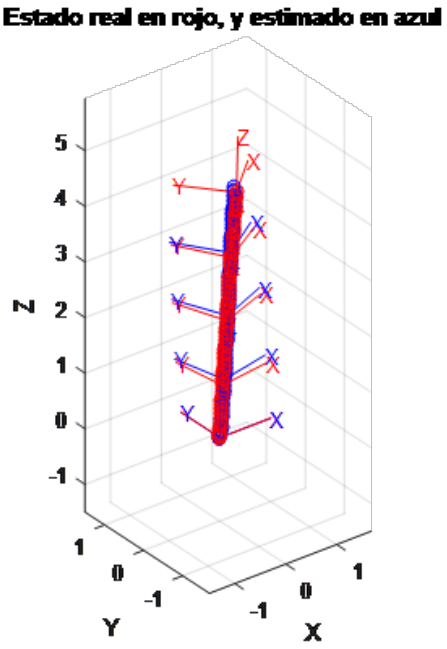


Figura 4.37: Etapa de Predicción del filtro prueba 1: Medidas de la IMU y el Láser.

Se han realizado diferentes pruebas para comprobar el funcionamiento del EKF. A continuación se muestran las pruebas realizadas con los resultados obtenidos de las diferentes configuraciones.

Configuración 2: Avance en el eje Y.

Tras la realización de la primera prueba, se realiza una segunda prueba introduciendo un comando de velocidad en *Roll* (Φ) en la señal de control u, siendo la señal de control $u = [1\ 0\ 0\ 0]'$. Los resultados obtenidos en distintos casos son los siguientes:

- **Etapa sin Predicción del filtro: Sólo medidas del láser**

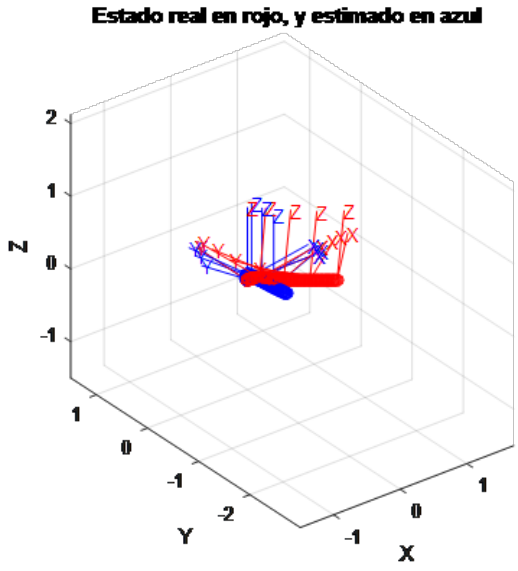


Figura 4.38: Etapa de Predicción del filtro prueba 2: Sólo medidas del láser.

- **Etapa sin Predicción del filtro: Sólo medidas de la IMU**

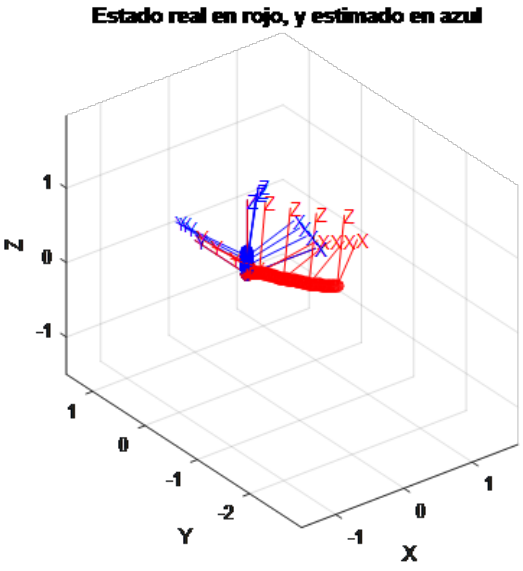


Figura 4.39: Etapa de Predicción del filtro prueba 2: Sólo medidas de la IMU.

- **Etapa sin Predicción del filtro: Medidas de la IMU y el Láser**

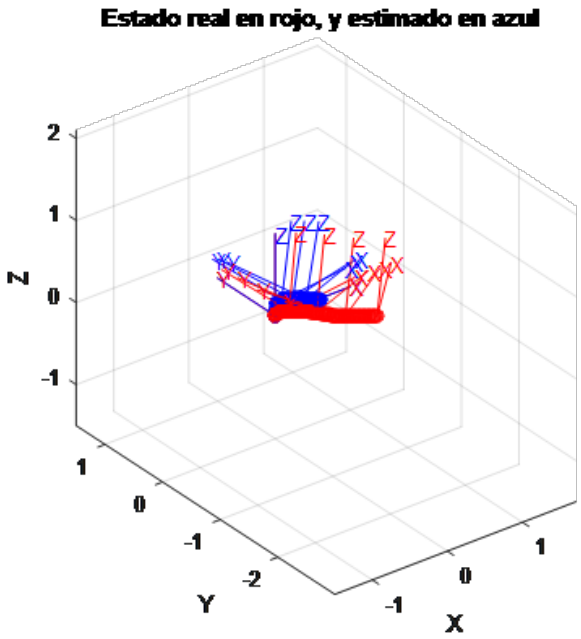


Figura 4.40: Etapa sin Predicción del filtro prueba 2: Medidas de la IMU y el Láser.

- **Etapa de Predicción del filtro: Sin medidas**

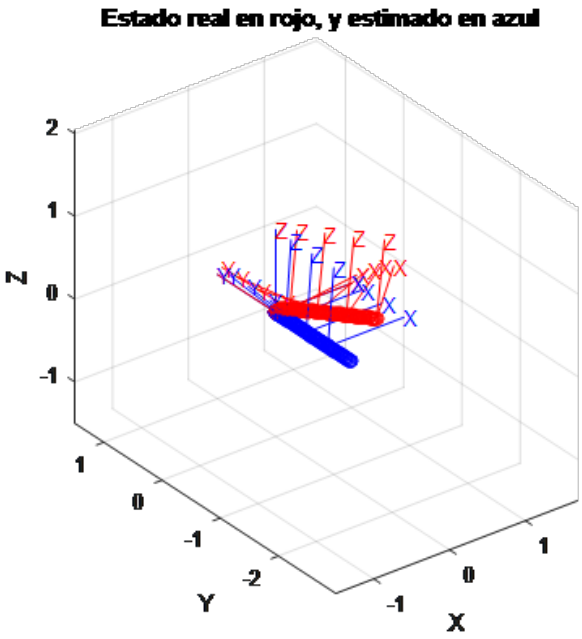


Figura 4.41: Etapa de Predicción del filtro prueba 2: Sin medidas.

- **Etapa de Predicción del filtro: Sólo medidas del láser**

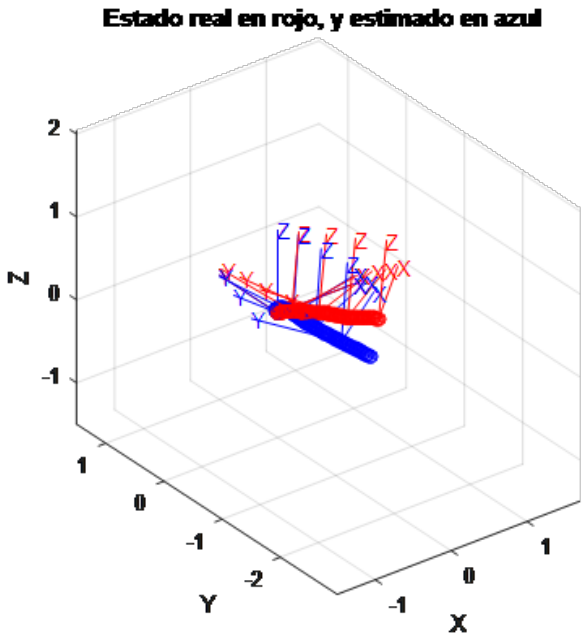


Figura 4.42: Etapa de Predicción del filtro prueba 2: Sólo medidas del láser.

- **Etapa de Predicción del filtro: Sólo medidas de la IMU**

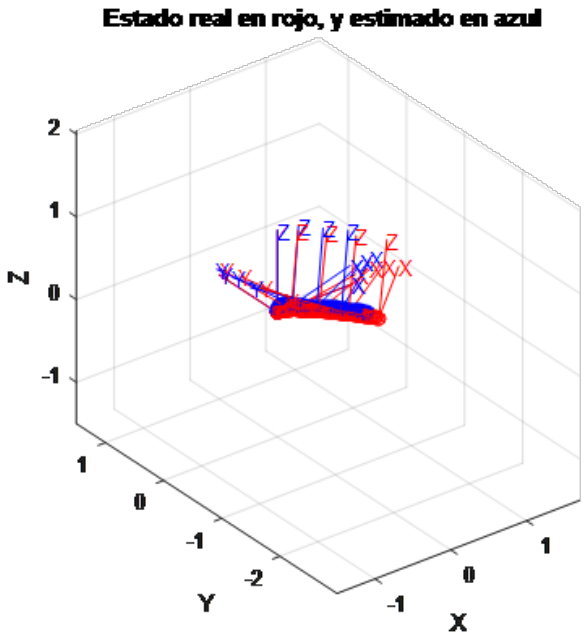


Figura 4.43: Etapa de Predicción del filtro prueba 2: Sólo medidas de la IMU.

- **Etapa de Predicción del filtro: Medidas de la IMU y el Láser**

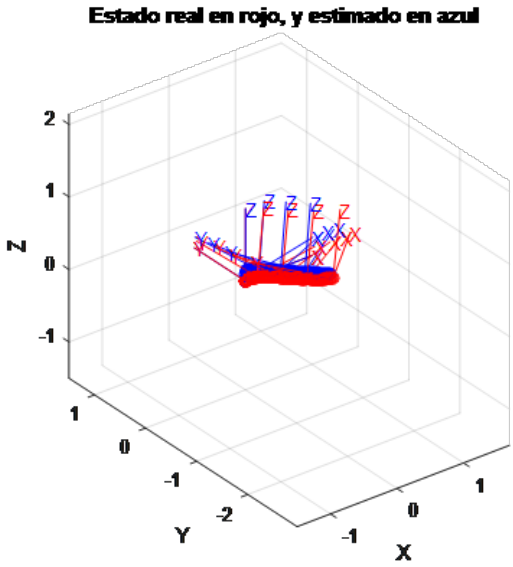


Figura 4.44: Etapa de Predicción del filtro prueba 2: Medidas de la IMU y el Láser.

Configuración 3: Avance en el eje X.

Otra prueba realizada se trata en un movimiento del drone a lo largo del eje X, se consigue mediante el envío de un comando de velocidad en *Pitch* (Θ) en la señal de control u, siendo la señal de control $u = [0\ 1\ 0\ 0]'$. Los resultados obtenidos en distintos casos son los siguientes:

- **Etapa sin Predicción del filtro: Sólo medidas del láser**

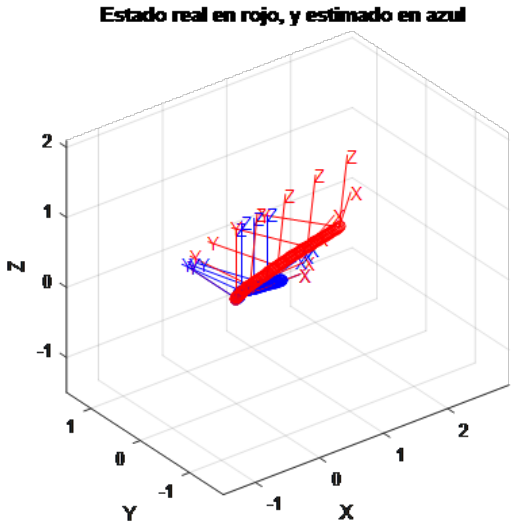


Figura 4.45: Etapa de Predicción del filtro prueba 3: Sólo medidas del láser.

- **Etapa sin Predicción del filtro: Sólo medidas de la IMU**

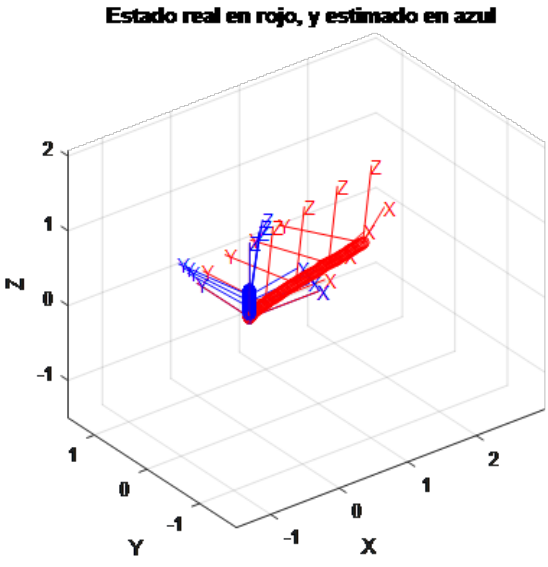


Figura 4.46: Etapa de Predicción del filtro prueba 3: Sólo medidas de la IMU.

- **Etapa sin Predicción del filtro: Medidas de la IMU y el Láser**

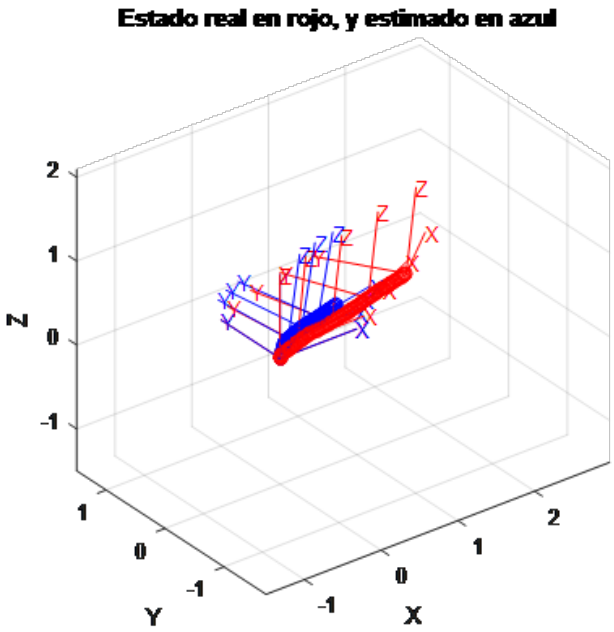


Figura 4.47: Etapa sin Predicción del filtro prueba 3: Medidas de la IMU y el Láser.

- **Etapa de Predicción del filtro: Sin medidas**

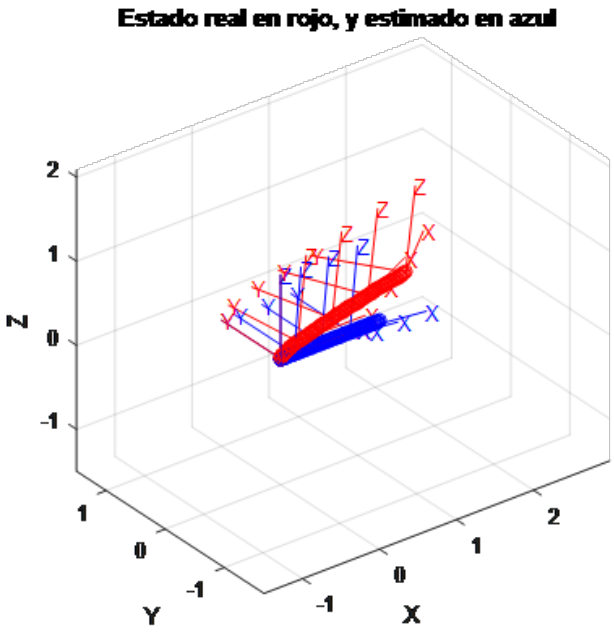


Figura 4.48: Etapa de Predicción del filtro prueba 3: Sin medidas.

- **Etapa de Predicción del filtro: Sólo medidas del láser**

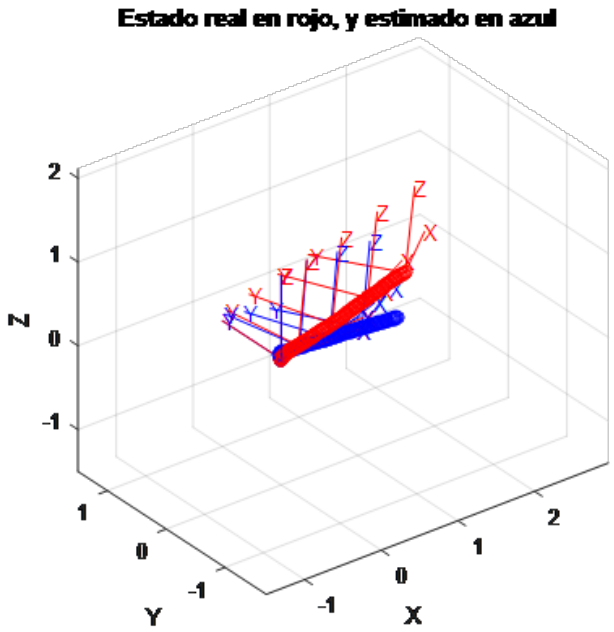


Figura 4.49: Etapa de Predicción del filtro prueba 3: Sólo medidas del láser.

- **Etapa de Predicción del filtro: Sólo medidas de la IMU**

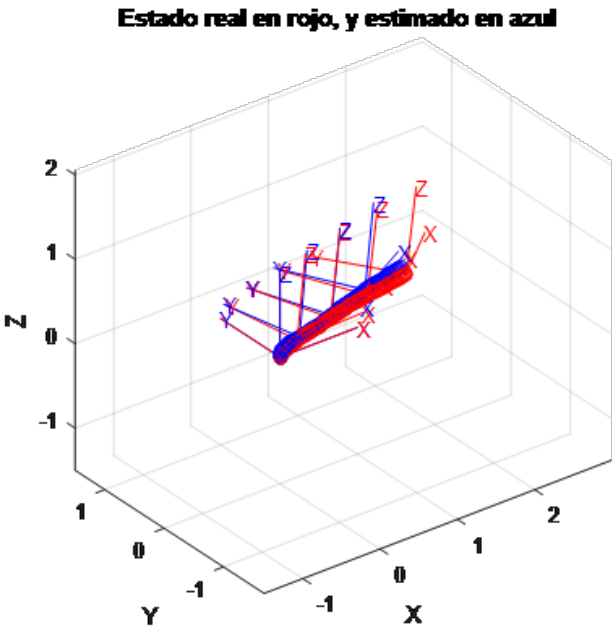


Figura 4.50: Etapa de Predicción del filtro prueba 3: Sólo medidas de la IMU.

- **Etapa de Predicción del filtro: Medidas de la IMU y el Láser**

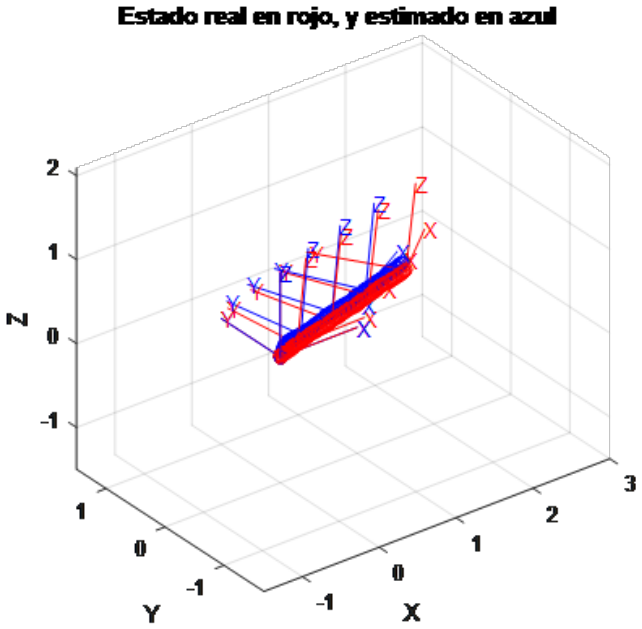


Figura 4.51: Etapa de Predicción del filtro prueba 3: Medidas de la IMU y el Láser.

Configuración 4: Rotación sobre el eje Z.

Por último, se realiza una prueba de rotación sobre el eje Z del drone y para ello se envía un valor en el ángulo Yaw en el comando en u, siendo la señal de control $u = [0\ 0\ 1\ 0]'$. Los resultados obtenidos en distintos casos son los siguientes:

- **Etapa sin Predicción del filtro: Sólo medidas del láser**

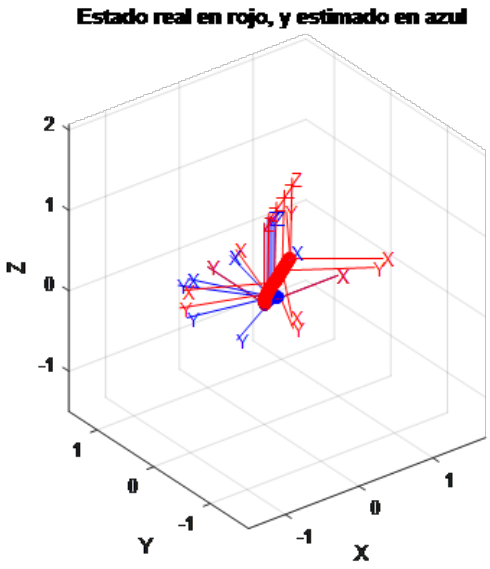


Figura 4.52: Etapa de Predicción del filtro prueba 4: Sólo medidas del láser.

- **Etapa sin Predicción del filtro: Sólo medidas de la IMU**

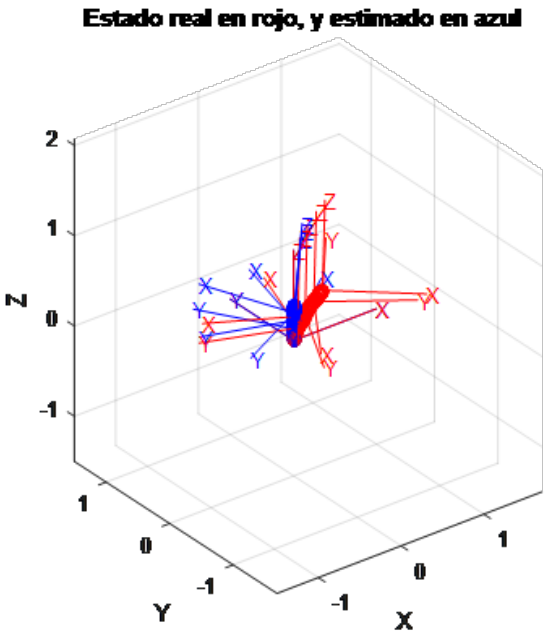


Figura 4.53: Etapa de Predicción del filtro prueba 4: Sólo medidas de la IMU.

- **Etapa sin Predicción del filtro: Medidas de la IMU y el Láser**

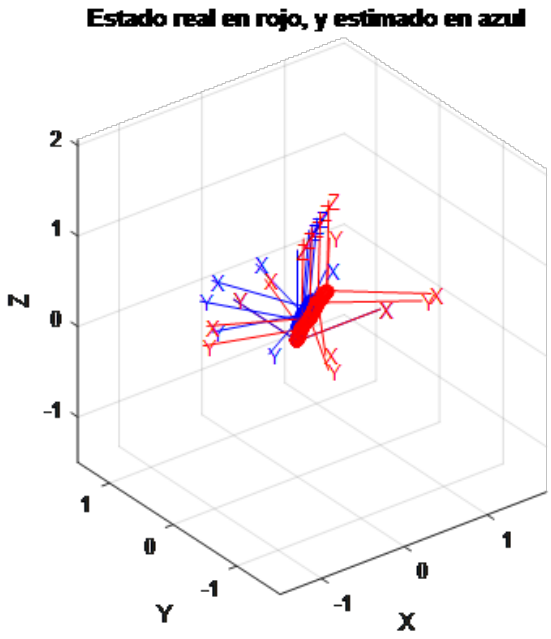


Figura 4.54: Etapa sin Predicción del filtro prueba 4: Medidas de la IMU y el Láser.

- **Etapa de Predicción del filtro: Sin medidas**

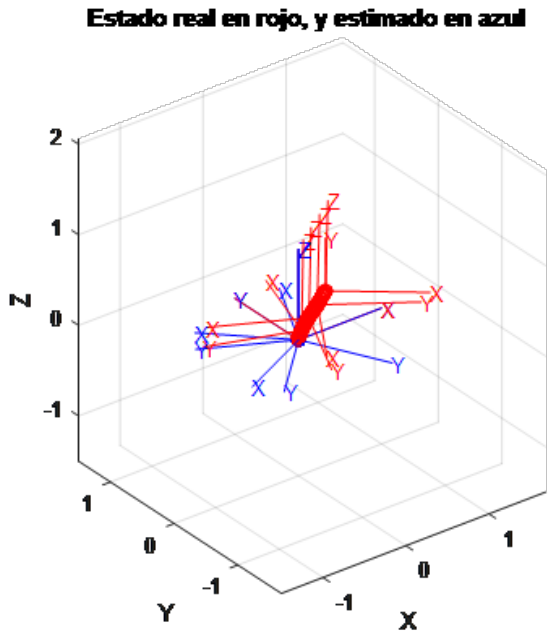


Figura 4.55: Etapa de Predicción del filtro prueba 4: Sin medidas.

- **Etapa de Predicción del filtro: Sólo medidas del láser**

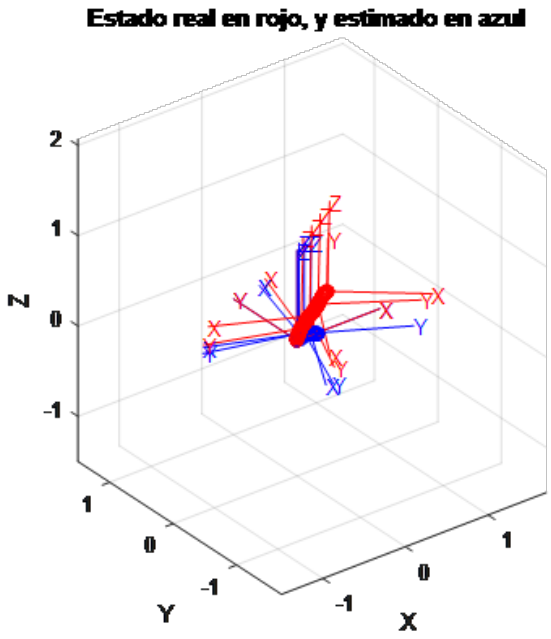


Figura 4.56: Etapa de Predicción del filtro prueba 4: Sólo medidas del láser.

- **Etapa de Predicción del filtro: Sólo medidas de la IMU**

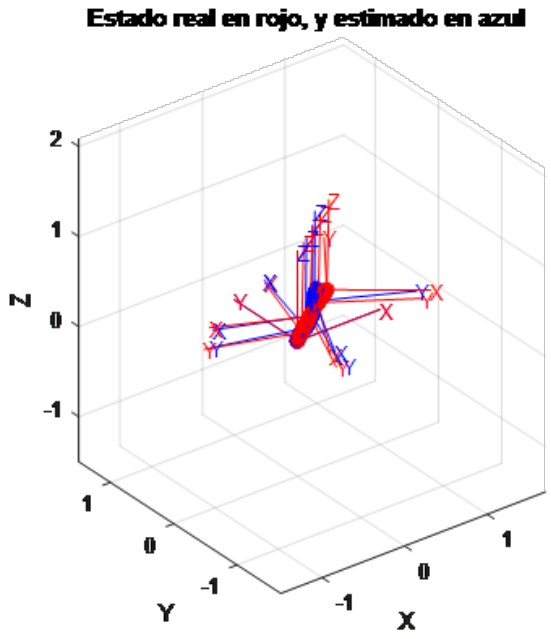


Figura 4.57: Etapa de Predicción del filtro prueba 4: Sólo medidas de la IMU.

- **Etapa de Predicción del filtro: Medidas de la IMU y el Láser**

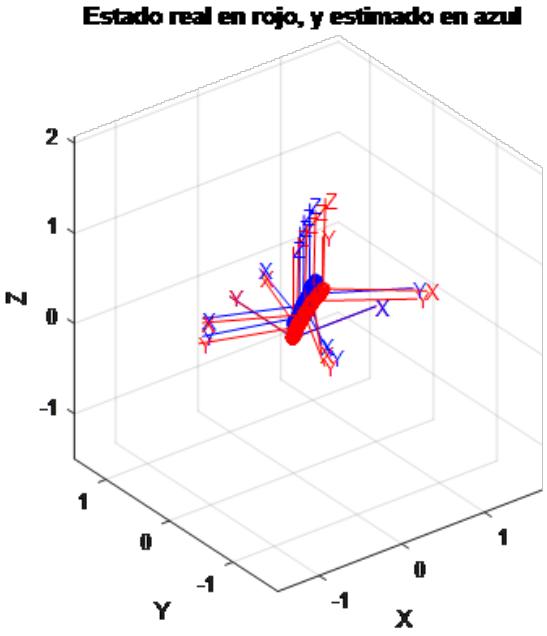


Figura 4.58: Etapa de Predicción del filtro prueba 4: Medidas de la IMU y el Láser.

4.3.3.4. Conclusión sobre el EKF

Como se puede observar en los resultados de las pruebas anteriores, el EKF creado predice con gran exactitud como se puede comprobar en la imagen 4.37. En los resultados obtenidos con el estimador de posición 3D basado en un EKF, se puede concluir que el uso de la etapa de predicción es necesaria ya que se obtienen resultados fiables. Como no se ha podido implementar en este proyecto el estimador de posición 3D que incorpora el algoritmo Hector-Mapping ni este estimador de posición 3D creado en MATLAB, esto provoca que los mapas obtenidos se aprecien pasillos recortados o más largo de lo normal ya que al no obtener información de la posición del robot, no se sabe exactamente la distancia real de los pasillos. Por ello, el uso del EKF permite obtener unas representaciones más precisas ya que consigue fusionar la información del láser con la IMU del MAV para obtener una posición más precisa y, por ende, un mapa 2D mejor.

Capítulo 5

Conclusiones y trabajos futuros

Para finalizar el proyecto, se agrupará en este apartado un resumen de las principales conclusiones a las que se ha llegado, comprobando también si se han conseguido lograr los objetivos marcados al inicio del documento. Por ultimo, se citará algunos de los trabajos futuros posibles que se pueden desarrollar como continuidad de este trabajo.

5.1. Conclusiones

Para comenzar con esta sección se analizará si se han conseguido los objetivos marcados en la sección 1.1 al comienzo de este proyecto. Para ello, se dividieron los objetivos en distintos apartados: conocimiento del entorno de desarrollo robótico ROS, simulación del algoritmo Hector-Mapping, implementación en la plataforma real de los sensores y del algoritmo SLAM, programación de un estimador de posición 3D y pruebas finales.

Como se ha visto en los apartados anteriores, se ha hecho un estudio del entorno de desarrollo robótico ROS del cual se han obtenido conocimientos necesarios para la realización de este proyecto. Una vez conocido el entorno de desarrollo, el siguiente paso ha sido el estudio en profundidad del algoritmo Hector-Mapping y conocer también cómo trabaja en ROS para poder obtener un entorno de simulación a partir de los paquetes proporcionados. Por lo tanto, los primeros objetivos de conocimiento se han logrado.

Tras obtener resultados coherentes en simulación y conocer bien el algoritmo en cuestión, a continuación se ha llevado a cabo el montaje de todos los sensores y los elementos necesarios sobre la plataforma róbotica (AR Drone 2.0) para la realización de este proyecto. Una vez montado todo, se ha llevado a cabo la implementación del algoritmo Hector-SLAM. Uno de los problemas encontrados aquí ha sido la limitación de peso para que el drone pueda volar correctamente ya que el AR Drone lleva un controlador interno específico para el peso sin elementos adicionales. Por lo tanto, se ha conseguido tener una estructura estable (dentro de lo que cabe), ya que durante periodos cortos de tiempo el AR Drone vuela correctamente, pero la limitación de peso hace que el vuelo no sea correcto del todo.

El algoritmo Hector-Mapping incorpora un estimador de posición 3D, este estimador no es posible implementarlo en el proyecto ya que la lectura de los sensores del drone no corresponde con el tipo de datos que el estimador necesita para estimar la posición, haciendo que la utilización de este estimador no sea recomendable. Se estudiaron otros paquetes de ROS que realizaban la misma función, como es el caso de *robot_pose_ekf*, este paquete sí necesita de odometría para la realización de la estimación de posición,

cosa que en este proyecto no utilizamos odometría ya que se trata de un drone y no un robot terrestre. Tras probar todas las posibilidades de estimadores 3D, se decidió realizar un algoritmo en MATLAB, que a partir de las medidas tomadas de los sensores del drone, estima la posición del drone. En definitiva, hubiera sido factible combinar el AR Drone 2.0 con el algoritmo Hector-Mapping para estimar la posición, pero aunque no obtengamos la posición 3D, se obtiene la posición 2D sobre el mapa y en combinación con el estimador creado, se puede obtener la posición 3D del drone.

5.1.1. Trabajos Futuros

Como se comentó en el apartado 2.1.3 el drone Bebop es un drone que tiene muchas posibilidades para remplazar a su predecesor AR Drone 2.0 ya que el Bebop cuenta con motores más potentes y puede aguantar mayor peso que el AR Drone 2.0. Así, los trabajos propuestos, teniendo en cuenta ambos drones, son:

- **Colaboración de los drones:** Un trabajo posible es la utilización del Bebop para que siga al AR Drone 2.0 para mejorar el SLAM, ya que ambos cuentan con cámaras incorporadas capaces de realizar este trabajo.
- **Sustitución del AR Drone 2.0 por el Bebop:** El Bebop es un drone que tiene muchas ventajas en este proyecto. Es pequeño por lo que puede entrar en casi cualquier lugar, aguanta mayor peso y, aunque no haya un paquete de drivers específicos en ROS para controlarlo, existen pequeños proyectos en ROS que permiten controlarlo con pocos comandos. Por lo que, un trabajo futuro propuesto es la realización de un paquete que permita controlar el Bebop de igual manera que el AR Drone 2.0 en ROS.
- **Utilización de la cámara de AR Drone 2.0 para realizar un SLAM 3D o mayor estabilización:** La cámara del AR Drone 2.0 puede dar mucha ventaja a la hora de realizar un SLAM 3D ya que está incorporada en la misma estructura del drone, por lo que no es necesaria una externa. Mediante la utilización de algunos paquetes que existen en ROS o en otras plataformas de desarrollo roboticas, se puede conseguir utilizar la cámara del drone para obtener un mapa 3D. Por otro lado, también puede ser utilizada para mejorar la estabilización del drone.

Como conclusión final, la realización de este proyecto ha sido buena ya que se ha conseguido muchos de los objetivos propuestos, y aunque algunos no se hayan conseguido, se ha buscado otra solución alternativa para llegar al mismo objetivo obteniendo como resultado una representación de un mapa en 2D con su respectiva estimación de posición.

MANUAL DE USUARIO

Manual de Usuario

En este capítulo se explica cómo se han de ejecutar los distintos programas y herramientas necesarias para la realización de este proyecto.

Para comenzar con este proyecto es necesario la instalación del entorno de desarrollo robótico ROS. Para ello es necesario visitar la página web *http://www.ros.org/* en ella se puede encontrar diferente información para entender en qué consiste ROS y para que es utilizado. La instalación se puede encontrar en el siguiente link (cada versión de ROS tiene su propio link y cada una tiene sus restricciones de software):

http://wiki.ros.org/ROS/Installation

Tabla 5.1: Link para la instalación de ROS.

Tras la instalación de ROS, es muy útil la utilización de los archivos *launch* ya que mediante estos archivos se pueden cargar varios paquetes a la vez sin necesidad de lanzar uno a uno. Para poder lanzarlos es necesario abrir primero un terminal y escribir *roscore*[1], este es el máster, y tiene que estar siempre en ejecución para que haya una comunicación entre los programas de ROS. La tabla 5.2 recoge los pasos necesarios para lanzar un arichivo *launch*, en este caso se trata de lanzar el ejecutable que simula el entorno en Gazebo del cuadricóptero de Hector.

>>Abrimos terminal y escribimos los siguientes comandos: >> $ roscore
>>Abrimos otro terminal y lanzamos el archivo roslaunch: >> $ roslaunch hector_quadrotor_demo indoor_slam_gazebo.launch

Tabla 5.2: Pasos para lanzar roscore y cargar un archivo launch.

La instalación de los paquetes es recomendable hacerla en un espacio de trabajo que normalmente se denomina *catkin_workspace*, mediante este espacio de trabajo se pueden instalar todos los paquetes que existen en el repositorio de ROS con la finalidad de poder acceder a todos los archivos del paquete por si en algún momento es necesario realizar una modificación en el mismo. Los pasos para crear un espacio de trabajo vienen en el siguiente link:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

Tabla 5.3: Link para la creación de un espacio de trabajo en ROS.

[1]Normalmente lanzado un archivo launch no es necesario lanzar previamente roscore ya que lo lanza a la vez que se carga el archivo launch

El link 5.3 explica los pasos necesarios para configurar el espacio de trabajo de ROS y cómo crear un entorno en el que se guarden todos los paquetes que se quieran utilizar. Este espacio de trabajo es muy útil ya que se puede crear tantos espacios de trabajo como se quiera dependiendo de los diferentes trabajos que se estén realizando en ROS.

Una vez creado el espacio de trabajo, es necesario instalar los paquetes que se vayan a utilizar. En el caso de este proyecto, el espacio de trabajo se ha denominado *tfg_ws*. A continuación se mostrará como se instala un paquete (para que sirva de ejemplo) en el espacio de trabajo:

- **Descargamos en archivo .zip el paquete:** En este caso se trata del paquete de *hector_quadrotor*[2].

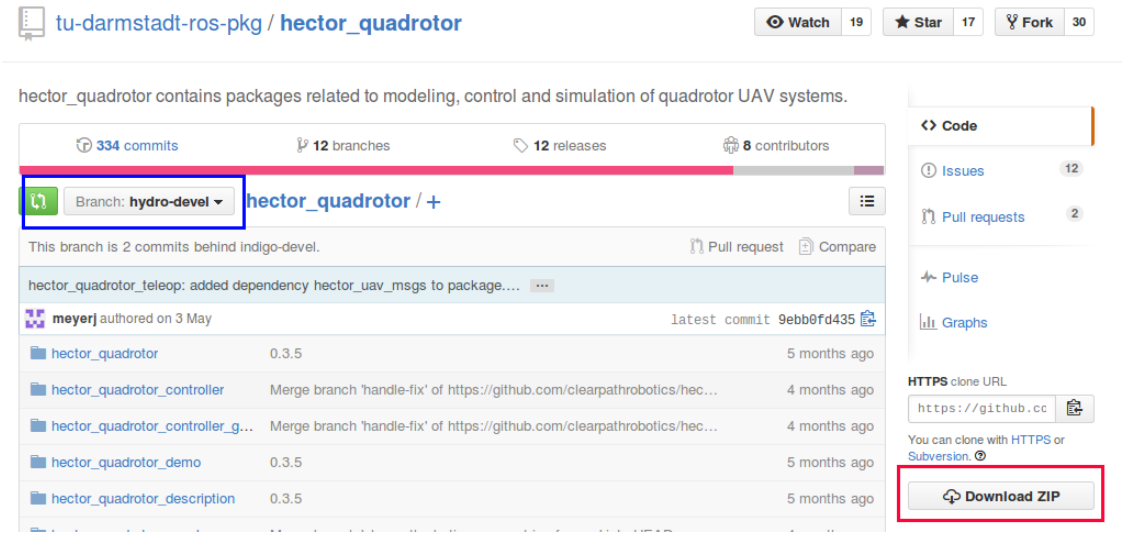


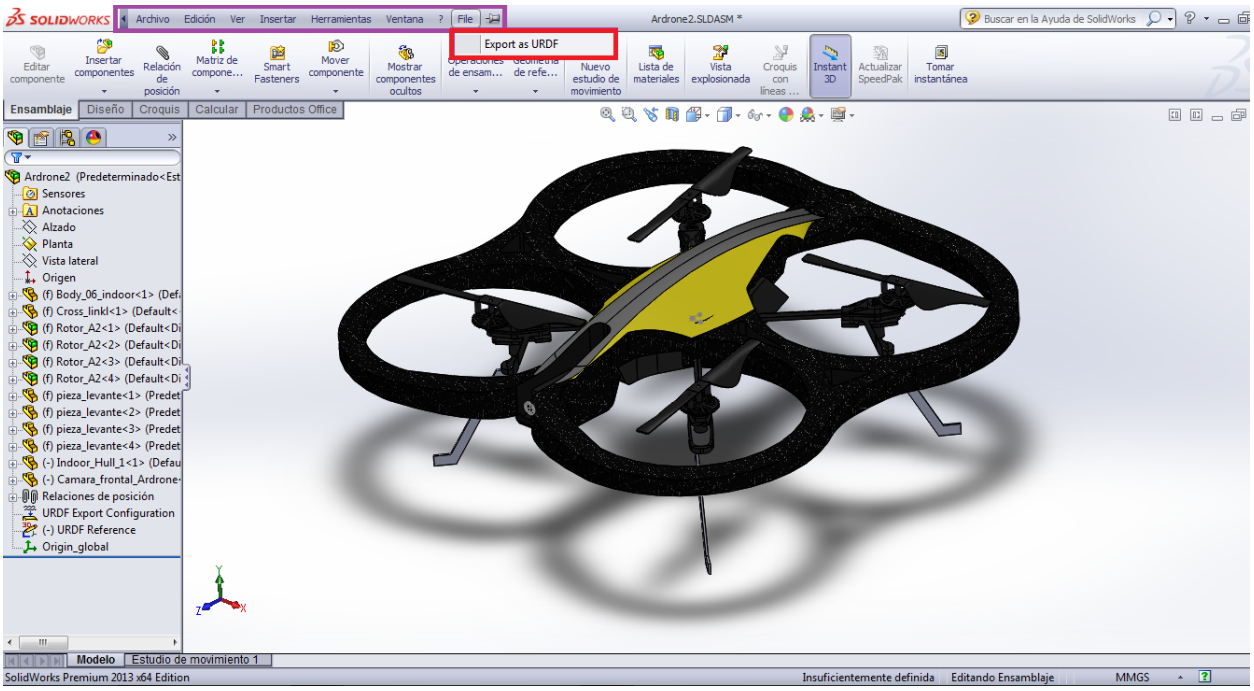
Figura 5.1: Descarga del paquete Hector. El recuadro rojo se sitúa sobre el botón para la descarga del paquete y el azúl hace recordatorio de la versión de ROS para la que funcionará dicho paquete.

- **Instalación del paquete:** Una vez descargado el paquete, se descomprime el fichero .zip en la carpeta source del espacio de trabajo tfg_ws/scr. Una vez copiado el paquete en esa carpeta, en un terminal nos situamos en el directorio tfg_ws e introducimos el comando "*catkin_make*. Si todo va bien se instalará el paquete, si hay algún error de dependencias de otros paquetes, será necesario instalar dichos paquetes.

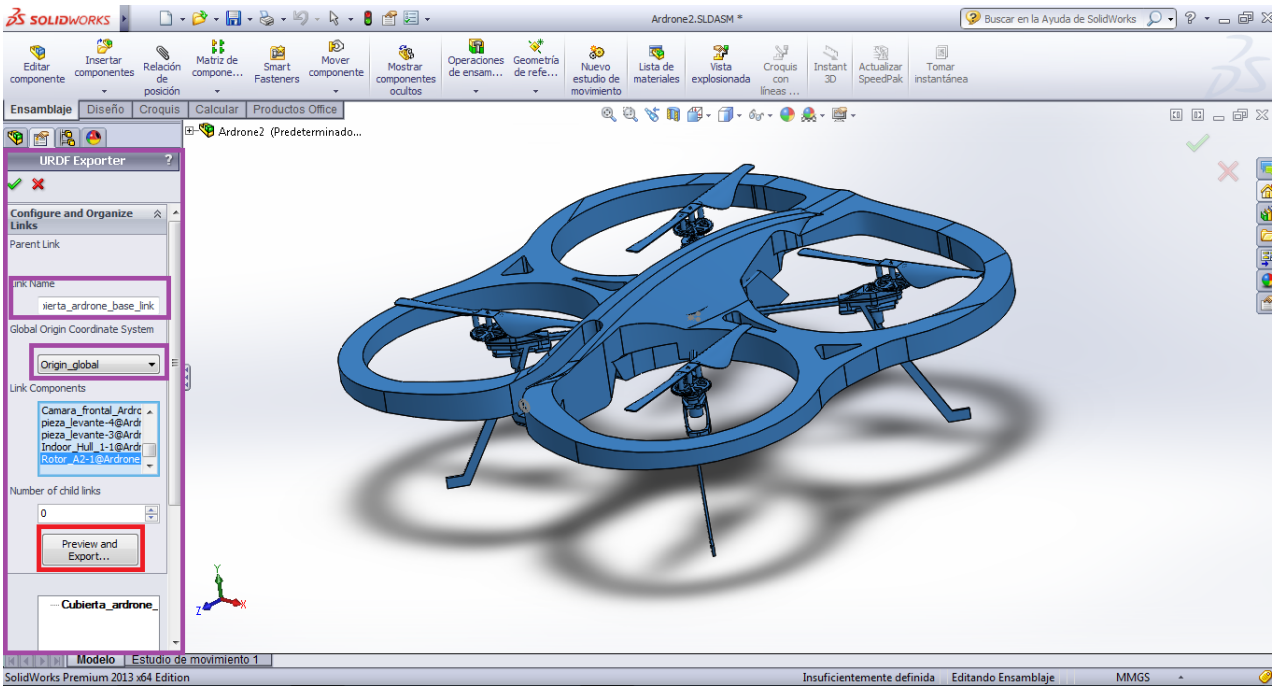
En este proyecto ha sido necesario la creación de un modelo 3D del AR Drone 2.0, este modelo ha sido creado en el programa *SolidWorks*. Como se comentó en el apartado 4.2, Gazebo utiliza un formato de lectura de modelos conocido como URDF, como en *SolidWorks* se obtienen los modelos en distinto formato, es necesario la utilización de una herramienta para poder pasar dichos archivos al formato URDF. Por ello, a continuación se explicará, una vez creado el modelo 3D en *SolidWorks*, cómo crear el archivo URDF.

Las siguientes imágenes explican, mediante recuadros de colores, dónde se tiene que pulsar para crear el paquete que contiene el modelo 3D en URDF:

[2]https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor



(a)



(b)

Figura 5.2: Pasos para obtener el modelo 3D en URDF

En la figura 5.2.(a) se muestran las pestañas (señaladas con recuadros de colores) que se deben pulsar para acceder al menú del *sw_urdf_exporter* y la figura 5.2.(b) muestra la ventana que se abre y los *links* que se han de seleccionar del robot para que se cree el archivo URDF. Una vez seleccionado todos los links, se ha de pulsar el recuadro rojo (*Preview and export*).

El siguiente paso se puede observar en la siguiente imagen:

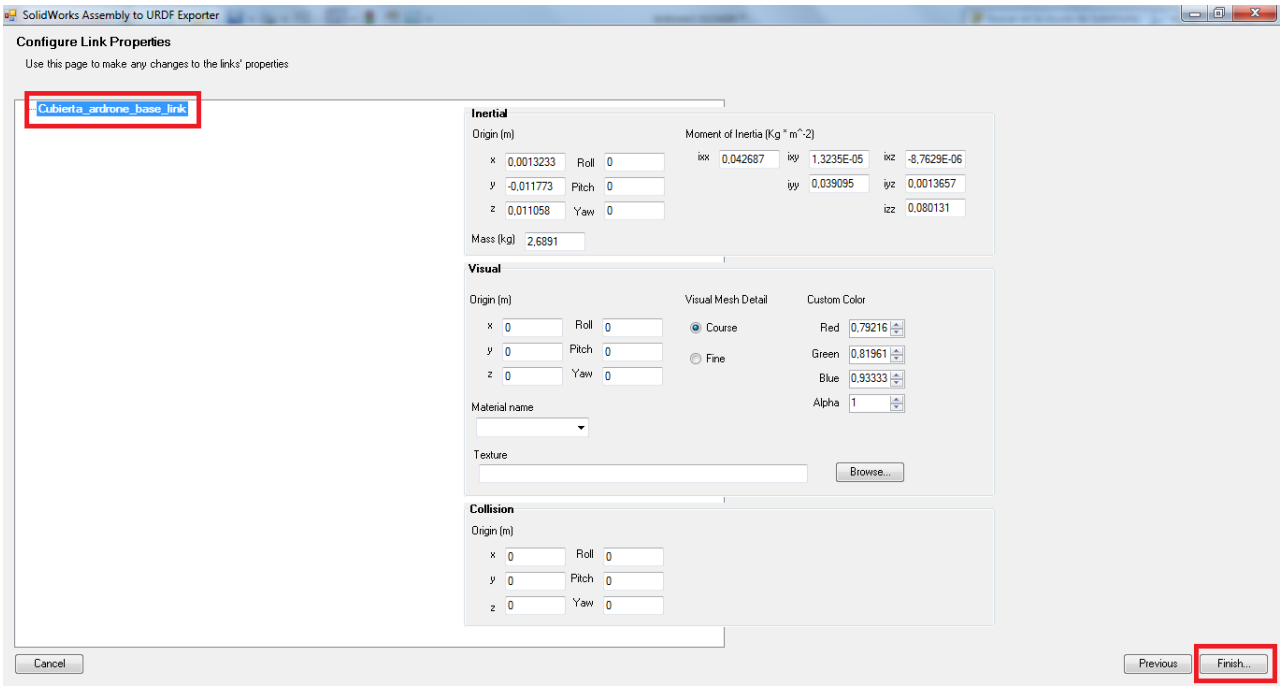


Figura 5.3: Paso final para la obtención del modelo 3D en URDF.

Al terminar se obtendrá una carpeta con diferentes archivos. Estos archivos pueden ser lanzados en Gazebo y Rviz para poder simular el algoritmo Hector-SLAM con el modelo 3D del AR Drone 2.0, ver figura 4.16.
Para ejecutar el algoritmo Hector-SLAM en la plataforma real, es necesario que el nodo del láser Hokuyo esté siempre ejecutandose[3]:

```
roslaunch hokuyo_node hokuyo_node
```

Robot real

Para realizar pruebas con el AR Drone es necasario conocer el paquete *ardrone_autonomy*, con este paquete se puede controlar el AR Drone y obtener la lectura de los sensores del drone. Para utilizar este paquete, el drone tiene que estar encendido y desde un terminal lanzar la siguiente línea de comando:

```
rosrun ardrone_autonomy ardrone_driver
```

Una vez lanzado el ejecutable que controla al AR Drone, se publican los topics que permiten obtener la lectura de los sensores y controlar el drone. Para visualizar las medidas de los sensores se realizan mediante el topic *ardrone_navdata*. Utilizando la siguiente línea de comando se puede visualizar en un terminal la lectura del topic *ardrone_navdata*:

```
rostopic echo /ardrone/navdata
```

[3]Es posible que los puertos del PC no estén abiertos para el láser por lo que hay que abrirlos con el siguiente comando: *sudo chmod a + rw /tty/ACM0*

Por otro lado, este ejecutable permite el control del AR Drone. Para su despegue es necesario publicar un mensaje vacío en el siguiente topic:

```
rostopic pub /ardrone/takeoff std_msgs/Empty " "
```

De igual manera se publica un mensaje para su aterrizaje:

```
rostopic pub /ardrone/land std_msgs/Empty " "
```

Para obtener más información sobre este paquete hace falta entrar en la siguiente página web:

```
http://ardrone-autonomy.readthedocs.org/en/latest/index.html
```

Estos pasos son los más utilizados en el proyecto y cada paquete funciona de la misma manera, por lo que hay que tener conocimiento de cada paquete y de cada ejecutable que lo forma.

PLANOS

Planos

Esquemáticos

En esta sección se muestran los esquemáticos de la plataforma robótica utilizada y del sensor láser Hokuyo.

Sensor láser Hokuyo

El láser utilizado en este proyecto ha sido el láser Hokuyo URG-04LX y sus dimensiones se muestran en la figura 5.4.

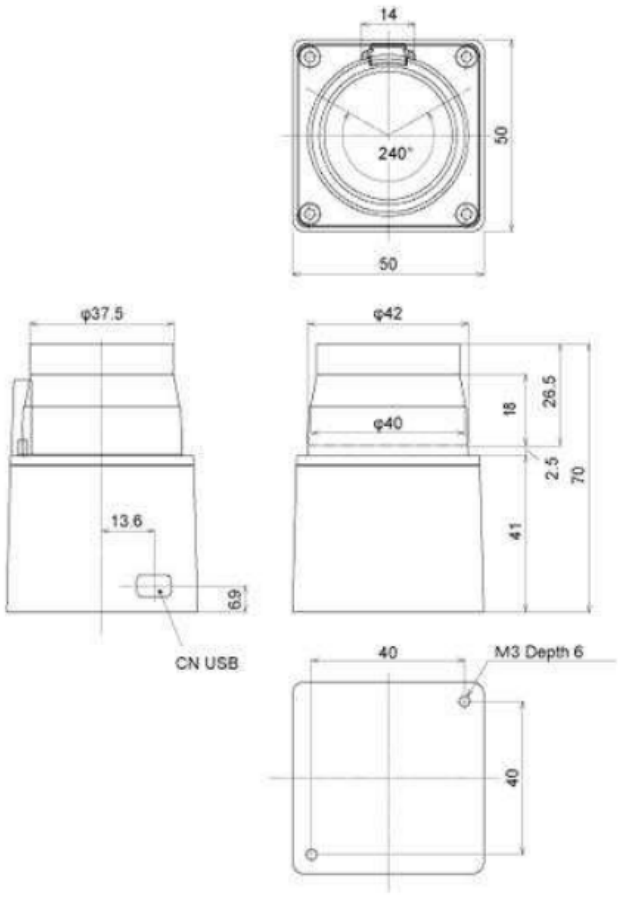


Figura 5.4: Planos del láser Hokuyo URG-04LX.

Las especificaciones del láser Hokuyo URG-04LX están mostradas en la figura.

Product name	Scanning Laser Range Finder
Model	URG-04LX
Light source	Semiconductor laser diode (λ=785nm), Laser power : less than 0.8mW Laser safety Class 1 (IEC60825-1)
Power voltage	5VDC ±5%
Power consumption	500mA or less (Start-up current 800mA)
Detection	60 mm ~ 4,095 mm (Guaranteed accuracy distance) 20mm ~ 5,600mm (Distance)* 4
Accuracy	Distance 20 ~ 1000mm: ±10mm* 2 Distance 1000 ~ 4000mm: ±1% of measurement* 2
Resolution	1 mm
Scan angle	240°
Angular resolution	0.36° (360° /1024)
Scanning speed	100msec/scan
Interface	RS-232C (19.2, 57.6, 115.2 ,500 ,750 kbps) USB Version 2.0 FS mode (12Mbps)
Ambient (Temperature/Humidity)	-10 ~ 50°C / 85%RH or less (without dew and frost)
Storage temperature	-25 ~ 75°C
Ambient light resistance	10000Lx or less
Vibration resistance	1.5mm double amplitude, 10 ~ 55Hz, X, Y and Z direction (2 hours), 98m/s² 55Hz ~ 150Hz in 2 minutes sweep, 1 hour each in X, Y and Z direction
Shock resistance	196 m/s², 10 times each in X, Y and Z direction
Protective structure	Optics : IP64 Case : IP40
Insulation	10MΩ for DC 500Vmegger
Weight	Approx. 160 g
Casing	Polycarbonate
Dimension (W×D×H)	50×50×70mm (Refer to design sheet No. C-40-3362)

*Under standard test conditions with white Kent sheet 70mm×70mm

Figura 5.5: Especificaciones del láser Hokuyo URG-04LX.

AR Drone 2.0

Las dimensiones del AR Drone 2.0 se pueden observar en la siguiente imagen.

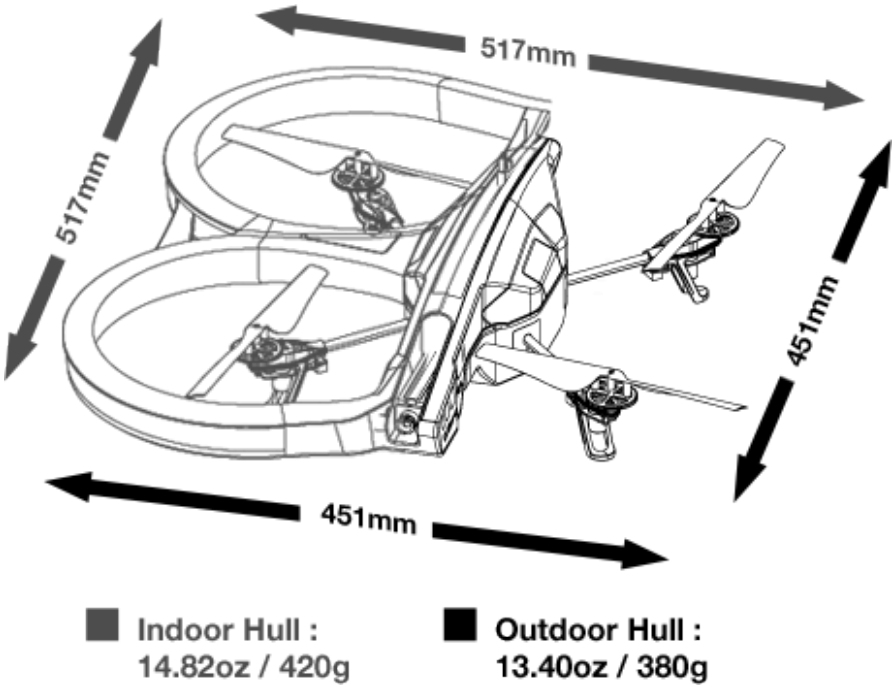


Figura 5.6: Dimensiones del AR Drone 2.0.

Programas

En esta sección se incluye un listado de los ejecutables y programas más utilizados en este proyecto, cuyo código detallado y comentado puede encontrarse en el CD que acompaña a esta memoria:

- Ejecutable para lanzar los drivers del Hokuyo: *hokuyo_node*.
- Ejecutable para lanzar los drivers del AR Drone: *ardrone_driver*.
- Archivo launch que lanza el algoritmo Hector-Mapping para simulación: *indoor_slam_gazebo.launch* del paquete *hector_quadrotor_demo*.
- Se ha creado un archivo que lanza el algoritmo Hector-Mapping para el robot real: *mapping_robot_real.launch*.
- Para el control del AR Drone desde teclado se ha creado un ejecutable denominado: *teleop_pr2_keyboard* partiendo como base del paquete *pr2 _teleop*.

PLIEGO DE CONDICIONES

Pliego de condiciones

Requisitos de Hardware

El hardware utilizado en este proyecto ha sido el siguiente:

- Ordenador compatible con PC. Cuanta mayor memoria RAM y capacidad de computo posea mejor funcionamiento tendrán los algoritmos.
- Es aconsejable tarjeta de aceleración gráfica si se desea trabajar con el simulador, puesto que el consumo de recursos es alto aunque el algoritmo Hector- SLAM utilice recursos bajos.
- Plataforma robótica AR Drone 2.0 y drivers necesarios para su control en ROS.
- Sensor láser Hokuyo URG-04LX con los drivers compatibles en el entorno de desarrollo ROS.
- Aunque no es de este proyecto, sí se ha utilizado una Raspberry Pi B+ y unos sensores de ultrasonidos.
- Dispositivo Android para poder controlar el AR Drone 2.0.

Requisitos de Software

El software necesario para este proyecto ha sido:

- Sistema operativo GNU/Linux, distribución Ubuntu, las pruebas y la programación de los algoritmos han sido realizadas en la versión 12.04 LTS.
- Sistema operativo Windows, el modelado de las piezas en 3D utilizadas han sido realizadas en este sistema operativo en la versión de Windows 7.
- Software de desarrollo robótico: ROS.
- Software de modelado 3D: SolidWorks.
- Software matemático: MATLAB

PRESUPUESTO

Presupuesto

Presupuesto

En esta sección se detalla el presupuesto de ejecución del proyecto y forma de las siguientes partes:

- Coste por uso de equipos y software.
- Coste de personal.
- Coste de materiales e impresión.
- Gastos generales, beneficio industrial e I.V.A.

La duración del proyecto ha sido de nueve meses ya que se engloba dentro de un proyecto de investigación del mismo tema. En este periodo de tiempo se incluye los tiempos de aprendizaje y de ejecución, y ha representado el trabajo de un Ingeniero Industrial a jornada parcial.

Coste por uso de equipos y Software

- Ordenador MacBook Pro con procesador Intel®Core^{TM}2 Duo CPU P8700 @ 2.53GHz, con 4GB de memoria RAM, de 64 bits y 250 GB de memoria ROM.
- Ordenador HP Pavilion dv6 de 64 bits y procesador Intel®Core^{TM}Duo i2, 4GB de memoria RAM y 500GB de memoria ROM.
- Impresora HP Laserjet 4100 Series PCL6.
- Software de libre distribución (tanto el sistema operativo GNU/LINUX como la plataforma de desarrollo ROS).
- Sistema operativo Windows 7.
- Software de modelado 3D SolidWorks®2013.
- Software matemático MATLAB.

Equipos	Precio Unitario (€)	Amortización	Uso	Total (€)
Ordenador MacBook Pro	1.299	4 años	9 meses	243.56
Ordenador HP i2	600	4 años	9 meses	112.5
Impresora HP Laserjet 400	400	4 años	1 mes	8.33
SO: Windows 7	18.97	Para siempre	4 años	18.97
Software: SolidWorks	55.92	Para siempre	3 meses	55.92
Software: MATLAB	149	4 años	4 meses	12.42
TOTAL:				451.7

Tabla 5.4: Presupuesto de Costes de Equipos y Software.

Coste de personal

Se detallan a continuación los sueldos base por hora trabajada de las personas que serían necesarias para realizar este proyecto, así como el coste que producirían las mismas. En este apartado se considera que se ha trabajado 20 horas semanales (jornada parcial) y se ha estimado que el mes tiene de media 22 días laborables. Por lo tanto el coste de personal ha sido:

Concepto	Horas	Precio por hora (€)	Total (€)
Ingeniero	792	29	22968
TOTAL:			22968

Tabla 5.5: Coste de personal.

Coste de materiales e impresión

El coste de materiales Hardware se refleja en la siguiente tabla:

Concepto	Precio (€)
AR Drone 2.0	299
Bebop	499
Hokuyo URG-04LX	1697
Raspberry Pi 2 B+	39,99
Ultrasonidos	5,99
Cables y conectores	8
TOTAL:	2548.98

Tabla 5.6: Coste de material Hardware.

Por otro lado, en la siguiente tabla se muestra el precio de coste del material de impresión:

Concepto	Precio (€)
Documentos para estudio y conocimiento del proyecto	10
Impresión del proyecto (gasto de tinta)	36
Impresión del proyecto (gasto de papel)	9
Encuadernación del proyecto	42
TOTAL:	97

Tabla 5.7: Coste de material de impresión.

Concepto	Precio (€)
Costes de material hardware	2548.98
Costes de impresión y encuadernación	97
TOTAL:	2645.98

Tabla 5.8: Coste total de material.

Presupuesto de ejecución de Material

En la tabla 5.9 muestra el coste total de todos los materiales, dicha suma total se denomina PEM.

Concepto de costes	Precio (€)
Equipos informáticos	451.7
Personal	22968
Materiales	2645.98
TOTAL:	26065.68

Tabla 5.9: Presupuesto de ejecución de material.

Presupuesto de ejecución por contrata

En el presupuesto de Ejecución por Contrata se incluye el coste de ejecución material (PEM) junto con los gastos generales, el beneficio industrial y los honorarios de dirección y redacción.

Concepto	Valor (% PEM)	Precio (€)
Presupuesto de ejecución de material (PEM)	100 %	26065.68
Gastos generales y beneficio industrial	15 %	3909.85
Honorarios de redacción	7 %	1824.59
Honorarios de dirección	7 %	1824.59
TOTAL:		33624.71

Tabla 5.10: Presupuesto de ejecución por contrata.

Presupuesto total

Tras haber expuesto todos los gastos que ha generado este proyecto reflejados en la tabla 5.10, al presupuesto total del proyecto hay que añadirle el IVA.

Concepto	Precio (€)
Presupuesto por ejecución de contrata	33624.71
IVA (21 %)	7061.19
TOTAL:	40685.9

Tabla 5.11: Presupuesto total del proyecto.

El presupuesto total del proyecto asciende a la cantidad aproximada de cuarenta mil seicientos ochenta y cinco coma noventa euros a fecha de septiembre del año 2015.

BIBLIOGRAFÍA

Bibliografía

[1] D. Mellinger and V. Kumar, *"Minimum snap trajectory generation and control for quadrotors"*. In: Proceedings IEEE Intelligent Conference on Robotics and Automation (ICRA). 2011

[2] Q. Lindsey,D. Mellinger and V. Kumar, *"Construction of cubic structures with quadrotor teams"*. In: Proceedings on Robotics: Science and Systems (RSS). 2011

[3] A. Kushleyev, D. Mellinger and V. Kumar, *"Towards a swarm of agile micro quadrotors"*. In: Proceedings of Robotics: Science and Systems (RRS). 2012

[4] R. Smith and P. Cheeseman, *"On the representation and estimation of spatial uncertainty"*. The international journal of Robotics Research. Massachusetts, vol. 5, 1986.

[5] Hugh Durrant-Whyte, *"Uncertain geometry in robotics"*. IEEE Robotics and Automation. vol. 4, 1988.

[6] T. Bailey and Hugh Durrant-Whyte, *"Simultaneous Localization and Mapping (SLAM): Part I"*. Robotics and Automation Magazine, Sydney, Australia, vol. 13, no 2, 1990.

[7] R. Smith, M. Self and P. Cheeseman, *"Simultaneous Localization and Mapping (SLAM): Part II State of the Art"*. Robotics and Automation Magazine, Sydney, Australia, vol. 13, no 3, 2006.

[8] S. Thrun, W. Burgard and D. Fox, *"A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots"*. Machine Learning and Autonomous Robots, vol. 31, 1998.

[9] S. Thrun, W. Burgard and D. Fox, *"Probabilistic Robotics (Intelligent Robotics and Autonomous Agents"*. The MIT Press, September, 2005.

[10] G. Welch, *"An 333 Inroduction to the Kalman Filter"*. University of North Carolina at Chapel Hill, vol. 17, 1995.

[11] S. Kohlbrecher, J. Meyer, O. von Stryk and U. Klingauf, *"A Flexible and Scalable SLAM System with Full 3D Motion Estimation"*. Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR), IEEE, November 2011.

[12] B. D. Lucas and T. Kanade, *"An iterative image registration technique with an application to stereo vision (darpa)"*. DARPA Image Understanding Workshop, April, 1981, pp 121-130.

[13] A. Burguera, Y. González and G. Oliver, *"On the use of likelihood fields to perform sonar scan matching localization,"* Autonomous Robots, vol. 26, pp. 203-222, May 2009.

[14] S. Grzonka, G. Grisetti and W. Burgard *"Towards a navigation system for autonomous indoor flying"*., In: Proceedings IEEE Intelligent Conference on Robotics and Automation (ICRA). 2009

[15] J. Engel, J. Sturm, D. Cremers, *"Accurate Figure Flying with a Quadrocopter Using Onboard Visual and Inertial Sensing"*. In: Proceedings of the Workshop on Visual Control of Mobile Robots (ViCoMoR)IEEE/RJS International Conference on Intelligent Robot Systems (IROS)., 2012.

[16] S. Moon, W. Eom and H. Gong, *"Development of a Large-Scale 3D Map Generation System for Indoor Autonomous Navigation Flight"*., In: Proceedings of Asia-Pacific International Symposium on Aerospace Technology, APISAT2014. 2014

[17] R. Li, J. Liu, L. Zhang and Y. Hang *"LIDAR/MEMS IMU Integrated Navigation (SLAM) Method for a Small UAV in Indoor Envir"*., In: Inertial Sensors and Systems. 2014

[18] S. Thrun, *"Robotic Mapping: A Survey"*., Carnegie Mellon University, Pittsburgh, PA: Morgan Kaufmann, 2002.

[19] S. Huang and G. Dissanayake, *"Convergence and Consistency Analysis for Extended Kalman Filter Based SLAM"*, Robotics, IEEE Transactions on, 2007, vol. 23.

[20] S. Maskell and N. Gordon, *"A tutorial on particle filters for online nonlinear/Non-gaussian Bayesian Tracking"*, In Proceeding of IEEE Colloquium on Tracking, 2002.